FireDAC avançado: Recursos de backup e restore, mestre-detalhe e mais

Clube Delphi

App Tethering na Prática
Compartilhe recursos e ações entre aplicativos

Behavioral Patterns
O comportamento de suas classes sob controle

Princípios SOLID
A solidez que seu sistema precisa

Edição 160 :: R$ 14,90

DEVMEDIA

AOP COM DELPHI

Orientação a Aspectos na Prática

ISSN 1517990-7

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??

NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

TENHA ACESSO A:

- +DE 290 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 9.074 VÍDEO-AULAS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

POR APENAS 69,90 MENSAIS

DEVMEDIA | Acesse: www.devmedia.com.br/mvp

Clube Delphi

160ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Joel Neto (joelrlneto@gmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Conteúdo sobre Arquitetura

04 – Orientação a Aspectos com Delphi
[*Filipe Dalepiane*]

Conteúdo sobre Novidades

13 – App Tethering: conectando aplicações Delphi
[*Fabrício Hissao Kawata*]

Artigo no estilo Mentoring, Conteúdo sobre Arquitetura

21 – Behavioral Design Patterns em Delphi
[*Guinther Pauli*]

Artigo no estilo Solução Completa

31 – FireDAC: utilização avançada
[*Filipe Dalepiane*]

Conteúdo sobre Boas Práticas, Artigo no estilo Curso

42 – Princípios SOLID com Delphi – Parte 1
[*Filipe Dalepiane*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Orientação a Aspectos com Delphi

Aprenda como aplicar este novo paradigma de programação

O Delphi, desde o seu surgimento, se caracterizou por ser uma excelente ferramenta RAD (Rapid Application Development), graças a poderosos componentes presentes na VCL, como TClientDataSet, TDataSetProvider e TDataSource.

Porém, no desenvolvimento de sistemas de grande, porte onde existem grandes equipes de desenvolvimento, não podemos ficar limitados a esses componentes e sim, utilizar o desenvolvimento Orientado a Objetos, aproveitando que o Delphi oferece todos os pré-requisitos que uma linguagem que suporta este paradigma deve ter (abstração, encapsulamento, herança e polimorfismo) e utilizando boas práticas de programação, princípios da orientação a objetos e padrões de projetos.

Infelizmente, o Delphi deixa um pouco a desejar no que diz respeito a existência de sólidos Frameworks que auxiliam o desenvolvedor a implementar sistemas Orientados a Objetos.

Já no que diz respeito ao compilador, não há o que reclamar, já que o Delphi no decorrer dos anos, principalmente após a aquisição da ferramenta pela Embarcadero, vem a cada versão trazendo novos recursos que possibilitam ao desenvolvedor maneiras de contornar a falta da existência de alguns frameworks prontos para desenvolvermos sistemas puramente Orientado a Objetos.

Mas isso por outro lado traz um benefício, já que os desenvolvedores têm que conhecer a fundo a linguagem e desenvolver suas próprias classes e Frameworks básicos que possam auxiliar nas atividades do dia a dia. Quem conhece os recursos avançados da linguagem como Métodos Anônimos, Generics, Custom Attributes e RTTI, consegue driblar esta falta de Frameworks através da criatividade.

Problemas da orientação a objetos

Todo software que segue o paradigma Orientado a Objetos possui uma parte de seu código que é alheio à implementação de comportamento do objeto. Estes códigos são todos aqueles que servem para o desenvolvimento de funcionalidades secundárias, e que se encontram espalhados por toda a aplicação (crosscutting concern). Alguns exemplos destas funcionalidades secundárias são Controle de Permissões, Manipulação de Exceções, Log de dados, Controle de Concorrência.

Mesmo que estas funcionalidades secundárias dos sistemas sejam implementadas em objetos, as chamadas aos métodos destas classes ficarão espalhadas por todo o sistema, caracterizando um auto acoplamento entre as partes básicas e secundárias dos sistemas.

A Programação Orientada a Aspectos surge para encapsular e modularizar esse código, e pode ser encarada como uma extensão da Programação Orientada a Objetos. Nela, encapsulamos uma determinada funcionalidade secundária em uma nova unidade: um aspecto. Esta funcionalidade é implementada de forma que garantimos que a mesma será aplicada nos instantes corretos da execução de um programa, sem que outros objetos envolvidos sejam afetados, de forma totalmente transparente.

Vale lembrar que a Orientação a Aspectos não pode existir de forma isolada, ela foi criada para trabalhar em conjunto com a

Fique por Dentro

Neste artigo aprenderemos como utilizar a orientação a aspectos em Delphi. Veremos todos os conceitos teóricos necessários para o perfeito entendimento desse paradigma de programação relativamente novo, e como ele vem complementar e aprimorar a programação orientada a objetos. Faremos um estudo completo de uma classe ainda pouco conhecida dos desenvolvedores Delphi, que é a TVirtualMethodInterceptor que permite a injeção de métodos antes, depois e ao ocorrer exceções em qualquer método virtual de qualquer classe que desejarmos, permitindo assim implementar um Framework básico de orientação a aspectos em Delphi. Serão utilizados vários recursos avançados da linguagem como Métodos Anônimos, Generics, Reflection e RTTI.

Orientação a Objetos, que ao contrário da primeira, pode existir de forma isolada.

A Orientação a Aspectos nos diz que toda a funcionalidade que não se refira a execução de regras de negócio, não deve estar presente na implementação do método e que todas as funcionalidades secundárias devem ser injetadas e executadas em outro lugar.

Observe, por exemplo, a **Listagem 1**. Quase toda sua implementação acaba ferindo o Princípio da Responsabilidade Única (**BOX 1**), que é um dos pilares da Orientação a Objetos. Segundo a Orientação a Aspectos, a única linha que deveria existir no método ExecutaOperacao é a linha 6, que faz a execução das regras de negócio.

BOX 1. Princípio da Responsabilidade Única

O princípio da Responsabilidade Única de classes indica que cada classe deve ter uma única responsabilidade, que deve estar totalmente encapsulada por ela. Assim, todos os seus métodos devem agir em conjunto para executar sua funcionalidade, desta forma, ela possui uma única razão ou motivo para ser alterada. Isto não significa que uma classe deva ter somente um método, mas que tenha métodos que juntos cheguem a um determinado objetivo comum. Uma classe poderá ser responsável por uma funcionalidade que pode precisar de cinco métodos para resolver sua tarefa. Classes assim propagam menos mudanças para o restante do sistema no caso da necessidade de alteração.

Listagem 1. Exemplo da mistura de interesses dentro de um mesmo método

```
01 procedure ExecutaOperacao;
02 begin
03   // Verifica Autorizacao;            Funcionalidade Secundária
04   // Inicia Transacao;                Funcionalidade Secundária
05
06   // Executa Regras De Negocio;       Funcionalidade Principal
07
08   // Finaliza Transação               Funcionalidade Secundária
09   // Grava Logs de Execução           Funcionalidade Secundária
10 end;
```

Programação Orientada a Aspectos

A Programação Orientada a Aspectos é um paradigma de programação que introduz um novo conceito no desenvolvimento de sistemas: o aspecto, que possibilita aos desenvolvedores de sistemas separar e organizar o seu código de acordo com sua importância para o sistema (Separation of Concerns). Ela permite realizar o encapsulamento de funcionalidades que são ortogonais dos sistemas.

Mas de que forma uma funcionalidade ortogonal é encapsulada em um aspecto? Como sabemos, as funcionalidades de um sistema são implementadas em blocos de código, que manipulam dados. Em orientação a objetos, encapsulamos uma funcionalidade em um objeto, que contém métodos e atributos. Os métodos contêm o código que implementa a funcionalidade, os atributos e os dados manipulados pelos métodos. Em Orientação a Aspectos, também temos que encapsular uma funcionalidade. Ainda que ortogonal, essa também pode ser implementada na forma de blocos de código. Tais blocos são conhecidos como Advices e estão contidos em um aspecto.

O que se pode notar é que a Orientação a Aspectos representa para a Orientação a Objetos o que a Orientação a Objetos foi para a programação estruturada, ou seja, que tenha uma separação de interesses e isolamento de partes do Software a fim de aperfeiçoar a manutenção.

Na prática, o que ocorre ao desenvolvermos utilizando a Orientação a Aspectos é a inversão de controle (**BOX 2**), pois os aspectos é que irão conhecer os componentes dos sistemas onde irão atuar e não os componentes que conhecerão os aspectos.

A principal função da Programação Orientada a Aspectos é identificar os interesses transversais das classes e determinar em que ponto eles devem ser executados. Normalmente os interesses transversais ou funcionalidades secundárias são associados a recursos não funcionais (**BOX 3**).

BOX 2. Inversão de Controle e Injeção de Dependência

A Inversão de Controle (IOC – Inversion Of Control) se dá quando a sequência de execução de chamadas de métodos é invertida em relação a programação tradicional, ou seja, não é chamada explicitamente. Estas chamadas geralmente são delegadas a Frameworks que se encarregam de fazer as chamadas necessárias.

A Injeção de Dependência sugere que as dependências de uma classe, como uma conexão com banco de dados, sejam injetados na classe. Com isso, além de inverter a dependência, a classe não precisa se preocupar com o ciclo de vida das suas dependências (no exemplo da conexão, a classe não precisa abrir/fechá-la, apenas receber uma referência e utilizá-la).

BOX 3. Requisitos Não Funcionais

Requisitos não funcionais são os requisitos relacionados ao uso de uma aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, facilidade de manutenção e tecnologias envolvidas. O cliente não fala a respeito de requisitos não funcionais ao solicitar o desenvolvimento de um sistema, pois eles são características mínimas de um software de qualidade, ficando a cargo do desenvolvedor optar por atender esses requisitos ou não.

- Demonstram qualidade acerca dos serviços ou funções disponibilizadas pelo sistema. Ex.: tempo, o processo de desenvolvimento, padrões de projeto, etc.
- Surgem geralmente conforme a necessidade dos usuários, em razão de orçamento e outros fatores.
- Podem estar relacionados à confiabilidade, tempo de resposta e espaço nas mídias de armazenamento disponíveis.
- Caso ocorra falha do não atendimento a um requisito não funcional, poderá tornar todo o sistema ineficaz. Ex.: requisito confiabilidade em um sistema de controle de voos.

Conceitos na Programação Orientada a Aspectos

A seguir são listados os principais conceitos referentes à Programação Orientada a Aspectos:

• Joinpoints: representam pontos bem definidos na execução de uma aplicação e eventos de interesses do fluxo de execução. Quando o programa passa por aquele ponto, um aspecto pode agir naquele momento. Exemplos de Joinpoints: chamadas de métodos, exceções.

• Pointcuts: são usados para representar um conjunto de Joinpoints, pois podem acontecer muitas ocorrências de Joinpoints de um mesmo tipo;

• Advices: são os métodos chamados quando os Pointcuts são ativados. Advices são as implementações dos aspectos.

Fazendo uma analogia sobre estes conceitos, podemos considerar os Aspectos sendo Classes, os Advices como os métodos, Joinpoints os atributos e os Pointcuts seriam algo como Triggers (gatilhos) sobre os Joinpoints.

Vantagens e Desvantagens da Orientação a Aspectos

Existem várias vantagens ao utilizar o paradigma de Orientação a Aspectos, dentre as quais se destacam o tamanho do código da aplicação, uma vez que separa-se a codificação referente aos Aspectos, ficando o código comum a várias partes da aplicação contido nos aspectos.

Outra grande vantagem está na manutenção do sistema, uma vez que o aspecto está contido numa única unidade, sendo necessário modificar apenas aquele ponto, ao invés de ir modificando todo o sistema.

A reusabilidade é outro fator que motiva a adoção deste paradigma, pois ao criarmos um aspecto bem codificado, como um controle de transações, fica muito fácil aplicá-lo em qualquer sistema. Veremos na parte prática estas vantagens.

Apesar deste paradigma ser inovador no desenvolvimento de softwares e apresentar inúmeras vantagens, existem algumas desvantagens e dificuldades em sua adoção.

Uma das desvantagens da Orientação a Aspectos é o fato de tornar o sistema mais complexo, exigindo uma boa análise de sistemas e desenvolvedores de alta qualidade, que saibam aplicar corretamente a Orientação a Aspectos nos sistemas, além da dificuldade do Debug de aplicações que utilizam Aspectos. Outra desvantagem é a falta de metodologias para o paradigma e de ferramentas de desenvolvimento de Aspectos nativas.

Nova RTTI do Delphi

Toda classe deve saber quais são suas propriedades, métodos, campos e atributos. A reflexão no Delphi é executada pela RTTI, que é um Framework interno da ferramenta de desenvolvimento e que disponibiliza meios para obtenção de tais informações em tempo de execução.

Através da RTTI podemos obter todos os métodos e propriedades com seus valores, de uma respectiva classe instanciada. Podem ser tanto as desenvolvidas por nós, quanto as já existentes no Delphi, como TLabel ou TSQLConnection por exemplo. Através dessas informações, podemos utilizá-las da forma que for necessário, podendo até mesmo alterar seus valores (no caso de propriedades e atributos), usando RTTI.

O uso da RTTI está quase sempre ligado a Frameworks que venham a ser desenvolvidos para centralização e otimização dos sistemas. Estes, independente de tamanho, têm por objetivo trazer maior flexibilidade a um sistema e produtividade ao desenvolvedor. Temos vários exemplos, como Frameworks de validação e persistência de dados.

O Delphi sempre ofereceu recursos de reflexão, porém, estes eram limitados quando comparados com a reflexão existente no .NET ou Java. Mas isso até a versão 2009, pois, a partir da versão 2010 do compilador, a RTTI passou por uma grande atualização, trazendo várias classes novas e benefícios esperados pelos programadores Delphi. No decorrer do artigo vamos entender o que é reflexão e como ela foi implementada no Delphi. Também veremos como utilizar na prática estes recursos.

Algumas classes introduzidas:

• TRttiContext;
• TRttiType;
• TRttiProperty;
• TRttiMethod.

A Classe TVirtualMethodInterceptor

O Delphi não possui suporte nativo à Orientação a Aspectos, mas graças a uma poderosa classe adicionada desde a versão XE do IDE, podemos realizar uma implementação para a representação de Aspectos em Delphi. Esta classe é a TVirtualMethodInterceptor e está presente no namespace System.Rtti.

Esta permite que interceptemos métodos virtuais de qualquer classe, fazendo a injeção de código antes, depois e na ocorrência de exceções destes métodos. Isso é possível devido ao fato dela ser capaz de modificar a VMT (**BOX 4**) de outras classes. Veja os eventos suportados:

• OnBefore: evento disparado imediatamente antes da execução de um método virtual;
• OnAfter: evento disparado imediatamente após a execução de um método virtual;
• OnException: evento disparado ao ocorrer uma exceção durante a execução de um método virtual.

BOX 4. VMT

A VMT (Virtual Method Table – Tabela de Métodos Virtuais) é uma tabela de métodos virtuais, onde são armazenados ponteiros para todos os métodos virtuais declarados em uma classe, e todas suas ancestrais, além de ponteiros para outras tabelas. O compilador determina o método a ser executado em tempo de execução, procurando pelo tipo de objeto. As tabelas de métodos virtuais permitem uma rápida execução das chamadas a métodos. O principal inconveniente é que elas exigem uma entrada para cada método de cada classe descendente, mesmo que este não seja implementado na classe descendente. Para contornar este problema, quando existe uma hierarquia muito grande de classes, há a opção pela diretiva Dynamic que tem o mesmo efeito do Virtual.

A maneira como esta classe consegue fazer estas interceptações é bastante interessante. É criada uma classe derivada da classe registrada, para introspecções, e em tempo de execução, a TVirtualMethodInterceptor substitui todos os métodos virtuais da classe base. Isso é obtido através da criação de uma nova tabela de métodos virtuais e preenchendo a mesma com as respectivas chamadas aos métodos configurados nos eventos OnBefore, OnAfter e OnException.

Na unit System.Rtti verificamos a existência de três tipos anônimos declarados referenciando Procedures. Todos possuem alguns parâmetros, conforme mostra a **Listagem 2**. Já na classe TVirtualMethodInterceptor, temos a declaração de três propriedades dos

tipos anteriores, conforme mostrado entre as linhas 8 a 10 da mesma listagem.

Listagem 2. Eventos e tipos declarados anônimosda unit System.Rtti

```
01 TInterceptBeforeNotify = reference to procedure(Instance: TObject;
02   Method: TRttiMethod; const Args: TArray<TValue>; out DoInvoke: Boolean;
03   out Result: TValue);
04 TInterceptAfterNotify = reference to procedure(Instance: TObject;
05   Method: TRttiMethod; const Args: TArray<TValue>; var Result: TValue);
06 TInterceptExceptionNotify = reference to procedure(Instance: TObject;
07   Method: TRttiMethod; const Args: TArray<TValue>; out RaiseException:
       Boolean;  TheException: Exception; out Result: TValue);
08 property OnBefore: TInterceptBeforeNotify read FOnBefore write FOnBefore;
09 property OnAfter: TInterceptAfterNotify read FOnAfter write FOnAfter;
10 property OnException: TInterceptExceptionNotify read FOnException write
     FOnException;
```

Como esta classe não é um componente, não programamos diretamente em seus eventos, e sim, atribuímos Procedures a essas propriedades para que quando ocorram, chamem automaticamente estes métodos.

Os três métodos anônimos (**BOX 5**) possuem alguns parâmetros iguais e outros diferentes. Vejamos uma descrição de cada um deles:

• Instance: Os três eventos recebem este parâmetro, que representa o objeto que está sendo interceptado naquele momento;

• Method: Os três eventos também possuem este parâmetro que representa qual é o método que está sendo chamado naquele momento;

• Args: Lista de argumentos passados para o método virtual;

• DoInvoke: Este é um parâmetro de saída (out) presente apenas no evento OnBefore e serve para cancelarmos a execução do método original se assim desejarmos, passando false para esta propriedade. Para prosseguir com a execução deixamos em True que é o valor padrão.

• Result: Valor de retorno do método;

• RaiseException: Parâmetro existente somente no evento OnException que serve para inibirmos a propagação da exceção para quem a chamou, atribuindo false ou deixando o valor padrão (True) que faz com que a exceção seja propagada para o método chamador.

• TheException: Também só existe no evento OnException e representa o objeto de exceção que foi gerado pela implementação do método virtual.

Esta poderosa classe possui ainda dois métodos públicos chamados Proxify e Unproxify. Ambas recebem como parâmetro um objeto e servem para registrar e cancelar o registro de objetos que desejamos que tenham sua tabela de métodos virtuais modificadas pela TVirtualMethodInterceptor. Da mesma forma que Create e Free, é interessante contornarmos o que tiver no interior destas declarações por Try/Finally, afim de garantirmos a restauração da tabela de métodos virtuais original e não termos nenhum erro de acesso a memória.

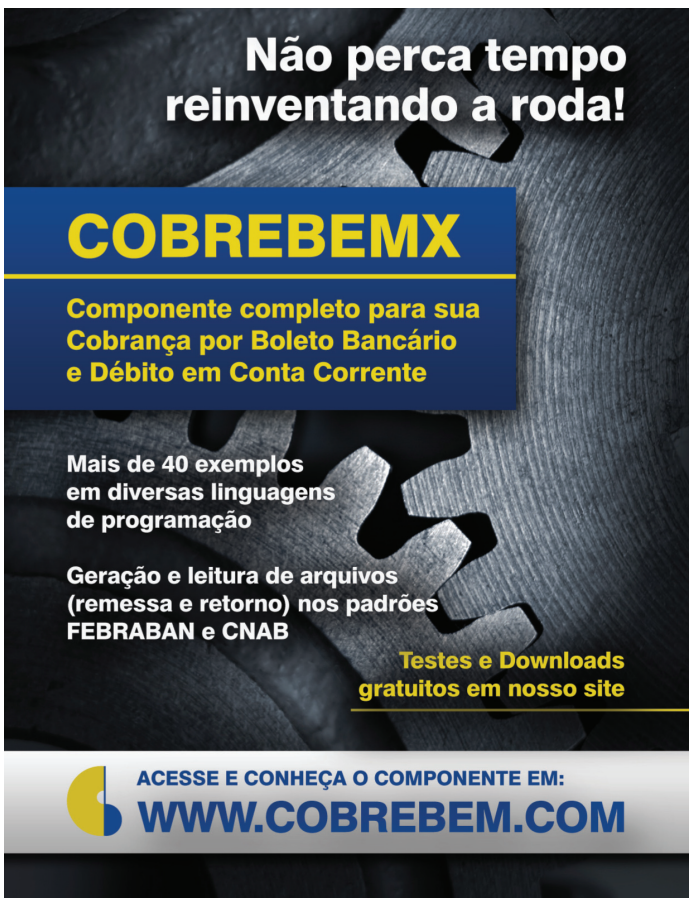
BOX 5. Métodos Anônimos

Métodos Anônimos é um recurso presente desde a versão 2009 do Delphi que permite a execução de métodos através de chamadas sem nome. Esses métodos podem ser atribuídos às variáveis, ponteiros ou ser passados como parâmetro. É ideal para quem trabalha com muitas iterações. A grosso modo, com métodos anônimos criamos referências para "alguma coisa" sem saber o que é. Métodos Anônimos podem ser definidos tanto como Procedures quanto Functions, além de poder possuir qualquer número de parâmetros ou nenhum. Podemos usar os já existentes na RTL do Delphi ou criarmos os nossos personalizados. Os tipos de Métodos Anônimos são declarados como referências para métodos (Procedures ou Functions).

Exemplo básico de uso

Como primeiro exemplo básico, criaremos uma classe TExemplo conforme mostrado na **Listagem 3.** Ela possuirá um método Executar que iremos utilizar para interceptar. Nela temos a declaração de uma classe simples para exemplificar a injeção de código antes, depois e ao ocorrer uma exceção, no método Executar desta classe. Este método apenas irá mostrar uma mensagem confirmando a execução do mesmo. Temos na linha 3 a declaração do método com a diretiva Virtual. Na implementação, temos apenas a exibição de uma mensagem (linha 10).

Note na **Listagem 3** que adicionamos propositalmente a declaração de uma referência para TExemplo (linha 8) e a chamada do método Executar (linha 11), sem que fosse feita a criação do objeto, para que assim, ocorra exceção neste momento, permitindo verificar o comportamento do programa durante a exceção.



Para fazer a interceptação de um método, a primeira etapa a ser realizada é declarar na seção private da unit principal o atributo FVMI do tipo TVirtualMethodInterceptor, conforme a **Listagem 4** (linha 7).

Listagem 3. Classe Exemplo

```
01 TExemplo = class(TObject)
02 public
03   procedure Executar(); virtual;
04 end;
05
06 procedure TExemplo.Executar;
07 var
08   E: TExemplo;
09 begin
10   MessageDlg('Execução do método', TMsgDlgType.mtInformation,
       [TMsgDlgBtn.mbOK], 0);
11   E.Executar;
12 end;
```

Listagem 4. Declaração Classe TFrmPrincipal

```
01 TfrmPrincipal = class(TForm)
02   btnExecutar: TButton;
03   procedure btnExecutarClick(Sender: TObject);
04   procedure FormCreate(Sender: TObject);
05 private
06   { Private declarations }
07   FVMI: TVirtualMethodInterceptor;
08 public
09   { Public declarations }
10 end;
```

Já na **Listagem 5** exibimos o evento OnCreate do formulário. Nele instanciamos a classe (linha 3), passando qual classe terá o monitoramento de métodos virtuais, configurando em seguida os três eventos mencionados anteriormente, OnBefore, OnAfter e OnException.

Neste exemplo da **Listagem 5**, apenas para o entendimento básico do uso da classe, apenas exibimos uma mensagem ao ocorrer a interceptação do método. Observe que o que atribuímos às propriedades entre as linhas 4 e 20 é um procedimento que deve ser executado, um Método Anônimo, e este deve possuir os mesmos parâmetros dos tipos declarados na unit System.Rtti, mostrados anteriormente.

No clique do botão Executar do formulário principal (**Listagem 6**), temos a criação de um objeto TExemplo (linha 5) e, através dos métodos Proxify (linha 7) e Unproxify (linha 11), fazemos o registro e cancelamento do registro do objeto que deverá ter sua tabela VMI modificada. Acrescentamos à chamada do método executar o try / finally, afim de garantirmos o cancelamento do registro do objeto no caso de ocorrer uma exceção.

Ao executar o aplicativo, é possível notar que o monitoramento realmente funciona, e percebemos que o mesmo abre um leque de possibilidades interessantes. Neste primeiro exemplo pode parecer um pouco trabalhoso ficar codificando os três eventos e ficar registrando as classes que o utilizam, porém, com um pouco de criatividade e conhecimento dos recursos disponibilizados pelo Delphi, veremos que é possível criar um Framework que facilita todo este trabalho.

Orientação a Aspectos em Delphi

Com a classe TVirtualMethodInterceptor podemos simular a Orientação a Aspectos no Delphi. Mas este recurso por si só não basta, nesta implementação utilizaremos vários outros recursos avançados da linguagem Delphi.

Listagem 5. Evento OnCreate do formulário principal

```
01 procedure TfrmPrincipal.FormCreate(Sender: TObject);
02 begin
03   FVMI := TVirtualMethodInterceptor.Create(TExemplo);
04   FVMI.OnBefore :=
05     procedure(Instance: TObject; Method: TRttiMethod; const Args:
       TArray<TValue>; out DoInvoke: Boolean; out Result: TValue)
06     begin
07       MessageDlg('Injetando código antes execução',
         TMsgDlgType.mtWarning, [TMsgDlgBtn.mbOK], 0);
08     end;

09   FVMI.OnAfter :=
10     procedure(Instance: TObject; Method: TRttiMethod; const Args:
       TArray<TValue>; var Result: TValue)
11     begin
12       MessageDlg('Injetando código após execução', TMsgDlgType.mtWarning,
         [TMsgDlgBtn.mbOK], 0);
13     end;

14   FVMI.OnException :=
15     procedure(Instance: TObject; Method: TRttiMethod; const Args:
       TArray<TValue>; out RaiseException: Boolean; TheException:
       Exception; out Result: TValue)
16     begin
17       MessageDlg('Injetando código em exceções', TMsgDlgType.mtError,
18         [TMsgDlgBtn.mbOK], 0);
19     end;
20 end;
```

Listagem 6. Evento OnClick do botão Executar

```
01 procedure TfrmPrincipal.btnExecutarClick(Sender: TObject);
02 var
03   Exemplo: TExemplo;
04 begin
05   Exemplo := TExemplo.Create;
06   try
07     FVMI.Proxify(Exemplo);
08     try
09       Exemplo.Executar();
10     finally
11       FVMI.Unproxify(Exemplo);
12     end;
13   finally
14     Exemplo.Free;
15   end;
16 end;
```

Nota

O Exemplo só poderá ser reproduzido a partir do Delphi XE, onde estão presentes todos os recursos utilizados e somente métodos com a diretiva Virtual poderão ser usados, por ficarem registrados numa tabela de métodos, onde o Delphi pode alterar dinamicamente.

Para auxiliar nesta tarefa, faremos o uso de Reflexão (**BOX 6**) em nossas classes. Em Delphi, fazemos o uso de Reflexão Computacional através de Custom Attributes. Um atributo permite que você defina informações adicionais sobre um método, propriedade ou classe que vão além das pré-existentes. Basta criarmos as classes herdando de TCustomAttribute que automaticamente a mesma já está apta a ser utilizada para anotar classes, conforme mostra a **Listagem 7**.

Neste exemplo definimos dois atributos, AccessControl que exemplifica o controle de acessos e TransactionControl que refere-se ao controle de transações.

BOX 6. Reflexão

Reflexão computacional (ou somente Reflexão) é a capacidade de um programa observar ou até mesmo modificar sua estrutura ou comportamento. Tipicamente, o termo se refere à Reflexão dinâmica (em tempo de execução), embora muitas linguagens suportem reflexão estática (em tempo de compilação). A reflexão é mais comum em linguagens de alto nível, como SmallTalk e menos comum em linguagens de mais baixo nível como o C.

Geralmente, Reflexão é a atividade que permite que um objeto obtenha informações sobre sua estrutura e sobre os motivos de sua computação. O paradigma de programação por trás dessa capacidade é chamado programação Reflexiva.

Quando o código fonte de um programa é compilado, a informação sobre sua estrutura normalmente se perde quando o código de baixo nível (tipicamente em assembly) é produzido. Já em um sistema que suporta reflexão, essa informação é preservada como Metadados, anexados ao código gerado.

Listagem 7. Unit uAttribute contendo as anotações que irão representar os aspectos

```
AccessControl = class(TCustomAttribute)
end;

TransactionControl = class(TCustomAttribute)
end;
```

A classe que fará o gerenciamento dos objetos que poderão ou não ser monitorados é a TAspect. Usamos um objeto da classe TObjectDictionary (**BOX 7**), que se encontra na unit Sytem.Generics.Collections, para criarmos dicionários de dados em memória através de chave/valor. Na **Listagem 8** temos a implementação da classe TAspect.

Note que na assinatura da linha 3 definimos que a chave será uma String**,** e que o valor contido no dicionário que será um objeto de TVirtualMethodInterceptor. Isto servirá para mantermos os objetos que queremos fazer a introspecção durante o ciclo de vida do programa. Usamos a diretiva "class var" para a declaração deste atributo, pois ele será um atributo de classe e não de instância, ou seja, não será necessário criar um objeto da classe TAspect para usá-lo. Os métodos públicos desta classe também são estáticos, para acessá-los utilizamos o nome da classe e não o nome do objeto (linhas 05 a 09). Além disso, teremos três métodos estáticos que servirão para registrar a classe que será monitorada e registrar e cancelar o registro de objetos. Veja que o método RegisterClass (linha 07) recebe um parâmetro do tipo TClass. Isto significa que podemos passar qualquer classe por parâmetro e não objetos. Já os métodos RegisterObject (linha 08) e UnregisterObject (linha 09) recebem como parâmetro um objeto do tipo TObject, ou seja, podemos passar qualquer objeto.

Por último, temos os métodos construtor e destrutor da classe TAspect, que também são estáticos e não servem para criar ou destruir objetos de TAspect. Estes apenas servem para fazer inicializações, conforme veremos adiante. Na **Listagem 9** temos a implementação dos métodos constructor e destructor da classe TAspect.

BOX 7. Dicionário de Dados

Dicionários de dados são estruturas de dados genéricas que representam pares de chaves/valores e são semelhantes a vetores, porém, o índice de um dicionário se chama chave. As chaves não devem ser repetidas e podem ser de qualquer tipo de dado. Estas estruturas também são chamadas de Arrays Associativos ou Maps. O Delphi utiliza Tabelas Hash para implementar Dicionários e Dados que otimizam a realização de pesquisas.

Listagem 8. Declaração da classe TAspect

```
01 TAspect = class(TCustomAttribute)
02 private
03   class var FVMIs: TObjectDictionary<string, TVirtualMethodInterceptor>;
04 public
05   class constructor Create;
06   class destructor Destroy;
07   class procedure RegisterClass(AClass: TClass);
08   class procedure RegisterObject(AObject: TObject);
09   class procedure UnregisterObject(AObject: TObject);
10 end;
```

Listagem 9. Construtor e Destrutor da Classe TAspect

```
01 class constructor TAspect.Create;
02 begin
03   FVMIs := TObjectDictionary<string,
     TVirtualMethodInterceptor>.Create([doOwnsValues]);
04 end;

05 class destructor TAspect.Destroy;
06 begin
07   FVMIs.Free;
08 end;
```

Note que há a implementação dos métodos construtor e destrutor. No método Create instanciamos o dicionário de dados (linha 03), informando que a chave será a String e o valor será um objeto de TVirtualMethodInterceptor, usando o parâmetro extra doOwnsValues ao criar o dicionário, que significa que terá valores nas entradas de dados. Já no Destroy, apenas liberamos o recurso da memória, conforme a linha 07.

Continuando a implementação da classe TAspect, na **Listagem 10** fazemos a codificação relacionada aos métodos RegisterObject e UnregisterObject.

O método RegisterObject primeiramente chama o método RegisterClass que registra a classe caso ainda não a possua, passando como parâmetro qual o tipo da classe que deve ser registrada (linha 03). Em seguida, é utilizada a propriedade Items do dicionário

que espera uma String, e que representa o nome da classe, sendo esta a chave do dicionário (linha 04). O método Items retorna uma instância da classe TVirtualMethodInterceptor, que definimos ser o tipo de dados que gostaríamos de armazenar no dicionário. Por último, chamamos o método Proxify (também na linha 04) para registrar o objeto. Já o método UnregisterObject também recebe um objeto como parâmetro e deverá cancelar o registro do mesmo. Primeiramente declaramos uma referência auxiliar para TVirtualMethodInterceptor (linha 08), que utilizamos para recuperar um objeto existente no dicionário. A seguir, fizemos um teste passando o nome do objeto para verificar se o mesmo já possui uma referência através do método TryGetValue (linha 10) que recebe uma chave no primeiro parâmetro e devolve um valor de saída no segundo, retornando também um tipo booleano caso haja sucesso na recuperação deste objeto. Havendo a referência, é chamado o método Unproxify (linha 12) que cancela o registro da introspecção.

Listagem 10. Registro e cancelamento do registro de TAspect

```
01 class procedure TAspect.RegisterObject(AObject: TObject);
02 begin
03   RegisterClass(AObject.ClassType);
04   FVMIs.Items[AObject.ClassName].Proxify(AObject);
05 end;

06 class procedure TAspect.UnregisterObject(AObject: TObject);
07 var
08   IVMI: TVirtualMethodInterceptor;
09 begin
10   if FVMIs.TryGetValue(AObject.ClassName, IVMI) then
11   begin
12     IVMI.Unproxify(AObject);
13   end;
14 end;
```

Por último, a **Listagem 11** ilustra o método RegisterClass da classe TAspect.

O método mais complexo e importante da classe TAspect é o RegisterClass da **Listagem 11**. Este recebe uma classe por parâmetro e serve para vincularmos uma classe a um objeto TVirtualMethodInterceptor do dicionário. Primeiramente, declaramos uma variável auxiliar chamada IVML (linha 03). Tentamos recuperar a referência desta classe do dicionário na linha 05, novamente utilizando o método TryGetValue. Se este objeto já existe no dicionário, nada é feito, pois o mesmo já estará registrado. Caso contrário, instanciamos um novo objeto de TVirtualMethodInterceptor, passando o tipo da classe que recebemos por parâmetro, afim de ser registrada, conforme a linha 07. Em seguida, temos a implementação do procedure (Método Anônimo) que será vinculado ao evento OnBefore (linha 09). Neste procedimento declaramos uma variável que referencia TCustomAttribute, conforme a linha 12. A seguir, percorremos todos os possíveis atributos existentes no método que recebemos por parâmetro, através do recurso "For In", conforme a linha 14. Na sequência, testamos se o atributo é do tipo de Controle de Acesso (linha 16), se sim, chamamos o método que verifica se o usuário possui acesso (linha 17). Na linha 18 verificamos se existe um atributo de Controle Transacional, se sim, chamamos o método responsável pela inicialização da transação, conforme a linha 19. O segundo procedure criado é o do evento OnException, que apenas verifica se existe um controle transacional e, havendo, chama o método de Rollback, conforme as linhas 29 e 30. O último evento codificado na listagem é o OnAfter, onde também é validada a existência de controle transacional, chamando o método Commit (linha 41). Encerrando, temos a adição do objeto IVMI no dicionário, juntamente com o nome da classe que será a chave para posterior

Listagem 11. Implementação do método RegisterClass

```
01 class procedure TAspect.RegisterClass(AClass: TClass);
02 var
03   IVMI: TVirtualMethodInterceptor;
04 begin
05   if not FVMIs.TryGetValue(AClass.ClassName, IVMI) then
06   begin
07     IVMI := TVirtualMethodInterceptor.Create(AClass);
08     try
09       IVMI.OnBefore :=
10         procedure (Instance: TObject;  Method: TRttiMethod; const Args:
           TArray<TValue>; out DoInvoke: Boolean; out Result: TValue)
11         var
12           IAttribute: TCustomAttribute;
13         begin
14           for IAttribute in Method.GetAttributes do
15           begin
16             if IAttribute is AccessControl then
17               TUsuario.VerificaPermissao(Method.Name);
18             if IAttribute is TransactionControl then
19               TConnection.BeginTransaction;
20           end;
21         end;
22       IVMI.OnException :=
23         procedure(Instance: TObject;Method: TRttiMethod;const Args:
           TArray<TValue>; out RaiseException: Boolean; TheException:
           Exception; out Result: TValue)
24         var
25           IAttribute: TCustomAttribute;
26         begin
27           for IAttribute in Method.GetAttributes do
28           begin
29             if IAttribute is TransactionControl then
30               TConnection.Rollback;
31           end;
32         end;
33       IVMI.OnAfter :=
34         procedure (Instance: TObject;  Method: TRttiMethod; const Args:
           TArray<TValue>;var Result: TValue)
35         var
36           IAttribute: TCustomAttribute;
37         begin
38           for IAttribute in Method.GetAttributes do
39           begin
40             if IAttribute is TransactionControl then
41               TConnection.Commit;
42           end;
43         end;
44       FVMIs.Add(AClass.ClassName, IVMI);
45     except
46       IVMI.Free;
47       raise;
48     end;
49   end;
50 end;
```

recuperação do objeto, conforme exibido na linha 44. A partir da linha 45 tem-se o bloco except que irá liberar o objeto IVMI caso ocorra uma exceção, propagando-a para o método que chamou, através da palavra reservada raise.

Nas próximas listagens temos a implementação das classes restantes, que como visto na **Listagem 11**. São elas as classes TConnection, TUsuario e TVenda, que serão abordadas a seguir. Primeiramente, temos a exibição da classe TConnection através da **Listagem 12**.

Observe que a classe TConnection é bem simples e possui apenas três métodos (BeginTransaction, Commit e Rollback, nas linhas 02 a 04). Como observado, entre as linhas 06 a 17 temos toda sua implementação. Os métodos são semelhantes, diferindo apenas no conteúdo da mensagem, como mostrado nas linhas 08, 12 e 16 respectivamente.

Continuando a explanação do conteúdo restante, temos a classe TUsuario ilustrada por meio da **Listagem 13**. Note que sua implementação é semelhante ao que foi realizado na classe anterior (TConnection), no entanto, temos apenas o método VerificaPermissao que recebe uma função como parâmetro. Para nosso exemplo, apenas uma mensagem que simula a permissão foi implementada na linha 06.

Listagem 12. Classe TConnection

```
01 TConnection = class
02   class procedure BeginTransaction;
03   class procedure Commit;
04   class procedure Rollback;
05 end;
06 class procedure TConnection.BeginTransaction;
07 begin
08   MessageDlg('Inicia Transação', TMsgDlgType.mtInformation,
       [TMsgDlgBtn.mbOK], 0);
09 end;
10 class procedure TConnection.Commit;
11 begin
12   MessageDlg('Commit', TMsgDlgType.mtInformation, [TMsgDlgBtn.mbOK], 0);
13 end;
14 class procedure TConnection.Rollback;
15 begin
16   MessageDlg('Rollback',TMsgDlgType.mtInformation, [TMsgDlgBtn.mbOK], 0);
17 end;
```

Listagem 13. Classe TUsuario

```
01 TUsuario = class
02   class procedure VerificaPermissao(AFuncao: string);
03 end;
04 class procedure TUsuario.VerificaPermissao(AFuncao: string);
05 begin
06   MessageDlg('Verifica Permissão',TMsgDlgType.mtInformation,
       [TMsgDlgBtn.mbOK], 0);
07 end;
```

Por último, finalizando nosso artigo, temos as **Listagens 14, 15 e 16** que referem-se à classe TVenda detalhadamente. Primeiramente temos a definição completa da classe TVenda na **Listagem 14**. Nela observamos dois métodos públicos, Salvar e SalvarComAspectos (linhas 08 e 11). O primeiro método utiliza o método tradicional de verificar permissões e trabalhar com transações, já o segundo faz o uso de atributos que decoram o método e indicam que ali haverá controle de acesso e controle transacional, conforme as linhas 09 e 10 respectivamente.

Na **Listagem 15** temos a implementação do construtor e do destrutor da classe TVenda. Note que o construtor faz uso da classe TAspect e do método RegisterObject (linha 03), para que seus métodos possam ser interceptados. Já o destrutor da classe chama o método responsável por cancelar o registro desta classe no monitoramento de métodos virtuais, conforme a linha 07.

Listagem 14. Declaração da classe TVenda

```
01 TVenda = class
02 private
03   procedure SalvarVenda;
04   procedure SalvarItens;
05 public
06   constructor Create;
07   destructor Destroy; override;
08   procedure Salvar; virtual;
09   [AccessControl]
10   [TransactionControl]
11   procedure SalvarComAspectos; virtual;
12 end;
```

Listagem 15. Contrutor e Destrutor da classe TVenda

```
01 constructor TVenda.Create;
02 begin
03   TAspect.RegisterObject(Self);
04 end;
05 destructor TVenda.Destroy;
06 begin
07   TAspect.UnregisterObject(Self);
08   inherited;
09 end;
```

Por ser registrado na criação e ter seu registro cancelado na destruição, durante todo o ciclo de vida de todas as instâncias criadas da classe TVenda, a classe TAspect fica livre para realizar toda e qualquer introspecção e modicar as chamadas de métodos virtuais da classe TVenda em tempo de execução. Por último, a **Listagem 16** aborda os métodos Salvar implementados de maneira tradicional e com Aspectos.

Ao analisarmos o método Salvar da **Listagem 16,** entre as linhas 01 a 13, percebe-se por sua declaração na listagem anterior que o mesmo não faz uso do conceito de Aspectos. Sendo assim, temos simplesmente a execução das regras de negócio, o restante, trata somente de funcionalidades secundárias e que segundo a Orientação a aspectos, não deveriam estar ali naquela implementação, mas sim em outra classe ou Framework que seriam os responsáveis por chamar esses métodos extras implicitamente, sem que o método Salvar precisasse se preocupar com essa tarefa.

Já na implementação do método SalvarComAspectos (linhas 14 a 18), temos somente a chamada aos métodos SalvarVenda e SalvarItens (linhas 16 e 17), ou seja, executando somente o que realmente deveria fazer. As funcionalidades secundárias

são injetadas automaticamente através dos Aspectos criados e com a utilização do conceito de Reflexão computacional, visto anteriormente.

Listagem 16. Método Salvar implementado com e sem aspectos

```
01 procedure TVenda.Salvar;
02 begin
03  TUsuario.VerificaPermissao('Salvar');
04  TConnection.BeginTransaction;
05  try
06   SalvarVenda;
07   SalvarItens;
08   TConnection.Commit;
09  except
10   TConnection.Rollback;
11   raise;
12  end;
13 end;

14 procedure TVenda.SalvarComAspectos;
15 begin
16  SalvarVenda;
17  SalvarItens;
18 end;

19 procedure TVenda.SalvarItens;
20 begin
21  MessageDlg('Salva Itens',TMsgDlgType.mtInformation,
    [TMsgDlgBtn.mbOK],0);
22 end;

23 procedure TVenda.SalvarVenda;
24 begin
25  MessageDlg('Salva Venda',TMsgDlgType.mtInformation,
    [TMsgDlgBtn.mbOK],0);
26 end;
```

A Orientação a Aspectos já deixou de ser promessa e já passa a ser parte do dia a dia de programadores de outras linguagens, como os desenvolvedores Java. Porém, seu uso em Delphi ainda não foi disseminado devido à baixa documentação de algumas classes do IDE.

Como vimos, apesar de não haver um Framework de Orientação a Aspectos, usando alguns recursos avançados da linguagem como Custom Attributes e Métodos Anônimos, conseguimos desenvolver um pequeno Framework que auxilia bastante no cotidiano de qualquer desenvolvedor.

A classe TVirtualMethodInterceptor ainda poderia ser útil no desenvolvimento de Frameworks de validações e de Injeção de Dependência. Enfim, as opções são muitas e como o Delphi ainda não possui Frameworks maduros e nativos para trabalharmos de forma avançada com a Orientação a Aspectos, cabe a nós desenvolvedores usar todos os recursos providos pelo compilador e criarmos nossas próprias classes e ferramentas para suprir essa falta.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi (Desktop) e C# (Web), utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Links:

Artigo Martin Fowler
http://www.martinfowler.com/bliki/InversionOfControl.html

Novidades do Delphi XE 6
https://www.embarcadero.com/br/products/delphi

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

App Tethering: conectando aplicações

Fique por dentro deste novo recurso

App Tethering é a denominação dada a um novo recurso introduzido na mais recente versão do Delphi, a XE6, cujo intuito principal é proporcionar uma interatividade mais direta entre as aplicações dos mais variados tipos, desenvolvidas sob o contexto do IDE. Seguindo pela perspectiva habitual da própria ferramenta, todo o panorama principal deste recurso é então refletido em componentes, disponíveis na Tool Palette do ambiente. Por esta razão, o uso destes se torna imprescindível para a plena incorporação do recurso na aplicação, o que viabiliza, neste momento, somente construções nativas. Em vista disso, a atividade inicial do artigo é contextualizar o leitor por meio de uma abordagem explicativa dos principais conceitos que permeiam o cenário "tethering" para, em seguida, ilustrar por meio de demonstrações reais a aplicação destes na prática.

No seu exercício cotidiano, uma aplicação Delphi provida de app tethering está habilitada a operar em três pontualidades essenciais. A primeira delas se dá pela localização efetiva de outras aplicações que também façam uso de app tethering, as quais estejam operando no mesmo contexto de execução. De forma específica, tal contexto se refere ao dispositivo ou aos dispositivos conectados, onde a "aplicação 1" está sendo executada juntamente das outras aplicações. O segundo ponto de operação deste tipo de aplicativo se dá pela execução de ações remotas, instantaneamente publicadas por uma determinada aplicação tethering e invocadas remotamente por outras. Por fim, seguindo por esta mesma linha, o terceiro ponto faz referência ao compartilhamento de dados e informações, na forma de recursos, entre as aplicações (VCL Forms e FireMonkey HD, VCL Forms e FireMonkey Mobile, etc.). Aqui, atualmente é provido o suporte tanto a tipos de dados (data types) tradicionais quanto a streams.

Em meio à tecnologia, como é de se esperar, todo cenário envolvendo interatividade requer determinados pontos de atenção no que diz respeito ao transporte de informações. No caso do recurso de tethering provido pelo Delphi, a resposta para toda sua interatividade citada é "nada em especial". Isto porque, tal artifício é suficientemente independente de qualquer meio de transporte ou protocolo específico. Nos bastidores, grande parte da incumbência destas questões, tal como a conexão em rede local (LAN), fica por conta da RTL, tornando-se transparente ao desenvolvedor. Em meio à comunidade, as aplicações Delphi que fazem uso do recurso de app tethering são tratadas simplesmente como "aplicações tethering", termo este que será adotado deste ponto em diante.

Fique por Dentro

Com o advento do recém-lançado Delphi XE6, novos recursos foram adicionados ao contexto da ferramenta. Logo, como já é de praxe a cada nova versão do IDE, tais novidades acabam por proporcionar novos desafios aos desenvolvedores, além de contribuir para um enriquecimento natural de todo o processo construtivo envolvido. Dentre os atrativos apresentados de forma exclusiva neste release está um recurso atrelado à RTL (Run-Time Library) denominado App Tethering que, em suma, provê a habilidade de interação entre aplicações Delphi, de qualquer segmento (Desktop ou Mobile), que partilhem de um mesmo ambiente local ou remoto. Na prática as probabilidades para este cenário são inúmeras, o que torna o aprendizado deste recurso de grande valia a partir deste momento.

Units

No Delphi, toda nova incorporação acaba por representar o acréscimo de novas entidades ao âmbito do IDE. Três dos casos mais corriqueiros se fazem pela adição de novas classes, novas units ou ambas, ao framework da ferramenta. Para o recurso de tethering, quatro é o número de units inclusas, cada qual com uma porção de classes relacionadas a um determinado fim. Nominalmente elas ficam sob o escopo System, na nova subdivisão Tether, conforme a seguir.

System.Tether.AppProfile

System.Tether.AppProfile.pas é o nome físico da unit que contempla, entre outras coisas, a declaração da classe TTethering-

AppProfile, que dá origem ao componente homônimo disponível no novo grupo Tethering da Tool Palette do IDE. No aspecto prático, seu uso é imprescindível para o ideal provimento do recurso de app tethering para uma aplicação, uma vez que irá representar o perfil (Profile) da mesma neste contexto. Em sua estrutura, há ainda a declaração de diversas outras classes, cada qual direcionada a um segmento específico e que, de forma direta ou indireta, estarão presentes durante toda a vida útil de aplicações tethering. A seguir são então listados alguns exemplos:

• TLocalAction: classe que representa uma determinada ação compartilhada a partir do ponto de vista do Profile detentor (local);
• TRemoteAction: semelhante a TLocalAction, porém, relacionado a um Profile remoto;
• TLocalResource: classe que representa um determinado recurso compartilhado a partir do ponto de vista do Profile detentor (local);
• TRemoteResource: semelhante a TLocalResource, porém relacionado a um Profile remoto.

System.Tether.Manager

Assim como System.Tether.AppProfile, a unit System.Tether.Manager também contempla a declaração da classe de um dos componentes disponíveis no novo grupo Tethering, aqui denominada TTetheringManager. Conforme pode se supor, sua representatividade perante a aplicação se dá na forma de um "gerenciador" específico para o recurso. Novamente, além dela, outras classes estão presentes nesta unidade de código, muitas das quais com o propósito de servir como base para outras classes atreladas ao contexto, tais como TTetheringProfile e TTetheringAdapter, que se estabelecem como ancestrais de TTetheringAppProfile e TTetheringNetworkAdapter, respectivamente. De forma complementar, é nesta unit que está declarada a classe de Exceção (descendente de Exception) pertinente ao uso dos recursos de tethering. A seguir é mostrada sua declaração inicial:

```
ETetheringException = class(Exception);
```

System.Tether.NetworkAdapter

Conforme sua própria denominação sugere, a unit System.Tether.NetworkAdapter agrupa os artefatos pertinentes às tratativas de rede de toda a parte de tethering. Em vista disso, duas são as classes declaradas em seu contexto: TTetheringNetworkManagerCommunicationThread e TTetheringNetworkAdapter. A primeira, relacionada essencialmente à linha de execução da aplicação, é uma especialização da classe base TThread (System.Classes.TThread) tradicional. Já a segunda, é voltada às tratativas de rede envolvidas neste cenário, onde atua como um "adaptador" para o aplicativo em questão, fornecendo todo o suporte de conexão a aplicações presentes no mesmo dispositivo ou na mesma rede local. Além de classes, a unit conta ainda com a declaração de alguns tipos (Types) permanentes, tal como TPairingState, que enumera os estados de paridade entre as aplicações – Waiting (aguardando), WaitingPairResponse (aguardando resposta), Paired (pareamento OK) e WaitingPassword (aguardando senha de acesso) – conforme pode ser notado em sua declaração a seguir:

```
TPairingState = (Waiting, WaitingPairResponse, Paired, WaitingPassword)
```

System.Tether.TCPProtocol

A nova unit System.Tether.TCPProtocol mantém a classe responsável por lidar com as condições de protocolo utilizadas na comunicação e troca de dados entre aplicações tethering. Aqui, o trabalho se dá sobre o protocolo de controle de transmissão padrão (TCP) sendo, portanto, direcionado a atuações em redes locais (LAN) e "in-device" (no próprio dispositivo). Nominalmente, a referida classe é apresentada como TTetheringTCPProtocol. Como requisito de uso, a classe trabalha com uma faixa de número de portas que vai de 2035 a 2100.

Componentes

Conforme já citado, este novo recurso de tethering introduzido no Delphi XE6 representa ao desenvolvedor, num primeiro momento, a disponibilidade de dois novos componentes – TTetheringManager e TTetheringAppProfile – localizados no também novo grupo "Tethering" da Tool Palette, conforme pode ser visto na **Figura 1**. Devido a este enxuto número de elementos, é correto afirmar que, na prática, o uso de ambos torna-se requisito essencial para a adição do recurso de "app tethering" a uma aplicação. Em vista disso, a seguir é apresentado um descritivo prático da função e usabilidade de cada um, em complemento ao que foi explicitado sobre suas classes anteriormente.

• TTetheringManager: uma aplicação Delphi irá necessitar deste componente para que ela seja capaz de descobrir outras aplicações que estão em execução no mesmo dispositivo ou dispositivos conectados, que também faça uso do recurso de app tethering. Tudo isso, a fim de proporcionar interligação e conectividade entre elas;
• TTetheringAppProfile: visando satisfazer o panorama proposto pelo recurso de tethering, este componente se torna de suma importância a uma aplicação, uma vez que é o responsável por definir as ações (actions) e os dados (data) que serão compartilhadas entre as aplicações interconectadas.

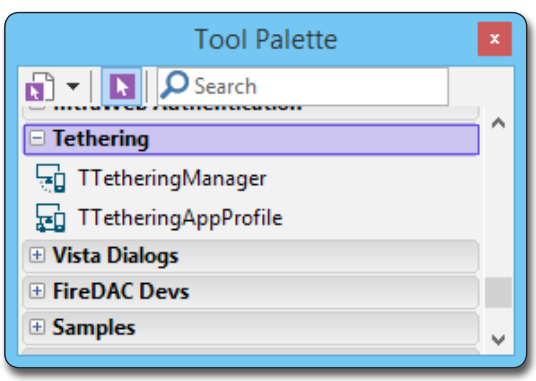


Figura 1. Grupo Tethering na Tool Palette do Delphi XE6

Conectividade entre aplicações "tethering"

Num primeiro momento o termo "tethering" pode parecer vago o suficiente para causar diversos questionamentos principalmente quanto ao seu propósito e/ou funcionalidade. Isso é proporcionado, de forma fundamental, pela própria simplicidade atrelada ao recurso que, em suma, tem a notória intenção de interligar aplicações e assim prover o compartilhamento de recursos entre elas. Até então, no cenário Delphi, propósitos similares a este poderiam (podem) ser atingidos de várias outras maneiras, utilizando-se as mais diferentes abordagens, sendo agora o uso de "tethering" mais uma dessas opções.

A ideia principal por trás do recurso, conforme já citado, é compartilhar ações e dados. Todavia, para que isso seja possível, o passo inicial se dá pela descoberta e posterior emparelhamento e conexão entre as aplicações. Neste cenário, há dois caminhos distintos a serem seguidos, ficando a escolha a cargo do desenvolvedor ou da exigência do próprio contexto atendido.

Conexão "automática"

A primeira abordagem, que pode ser considerada como a mais simples, envolve o uso de aplicações pertencentes a um mesmo "grupo" e emparelhadas por meio de uma abordagem denominada "automatically discover" (descoberta automática). Na prática, para que isso seja possível, dois são os requisitos necessários, ambos relacionados aos componentes envolvidos. O primeiro ponto diz respeito ao elemento de Profile (TTetheringAppProfile) pertencente à cada aplicação, mais especificamente à sua propriedade Group (**Figura 2**), do tipo string, que deverá receber a denominação comum para o grupo que será estabelecido. Já o segundo ponto, agora atrelado ao outro elemento envolvido (TTetheringManager), se faz pela chamada em runtime, em cada aplicação, ao método AutoConnect do componente. Em suma, tal método será o responsável pela descoberta e conexão "automática" das aplicações, tendo como base o grupo definido. Conforme pode ser visto logo a seguir, de acordo com sua definição, o mesmo recebe como parâmetro (Timeout) um valor Cardinal, que determina o tempo de duração do processo de "descoberta" citado.

```
procedure AutoConnect(Timeout: Cardinal = 1000);
```

No caso específico deste cenário, do ponto de vista comercial, a definição de um grupo tende a acarretar imprevistos, uma vez que sua especificação se dá pela informação de uma simples cadeia de caracteres (string). Um exemplo trivial seria a incidência da mesma denominação do grupo especificado em aplicativos de terceiros, acarretando eventuais emparelhamentos indesejados, ferindo diretamente questões de segurança. Logo, a fim de se estabelecer uma denominação exclusiva ao grupo, uma alternativa ideal se daria pelo uso de identificadores específicos à empresa ou mesmo pela adoção de identificadores UUID (Universal Unique Identifier **BOX 1**). No geral, tais abordagens são caracterizadas como "formas brandas" de proteção à conexão ao grupo de aplicações. Isto porque, do ponto de vista tecnológico, os aplicativos de terceiros ainda poderiam, de alguma forma, ter acesso à denominação do grupo, usufruindo desta questão para se conectarem intencionalmente às suas aplicações. A fim de se evitar ocorrências como esta, o recurso de tethering contempla uma forma mais "robusta" de proteção, refletida no uso de senhas de acesso, conforme será visto mais adiante neste artigo.

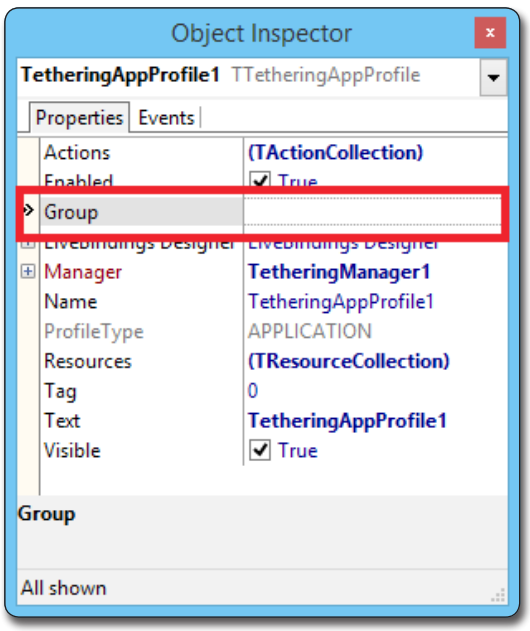


Figura 2. TTetheringAppProfile – propriedade Group

BOX 1. UUID

UUID é o acrônimo para Universally Unique Identifier que, sumariamente, pode ser definido como sendo um identificador universal exclusivo, utilizado de forma bastante concisa no provimento de softwares. Sua grande adoção se deve principalmente em razão de sua padronização chancelada pela OSF (Open Software Foundation). Do ponto de vista mais técnico, o UUID toma a forma de um número de 128 bits. Já sua representação visual é composta por 32 dígitos hexadecimais, apresentados em grupos, constituindo o formato 8-4-4-4-12, num total de 36 caracteres alfanuméricos.

Emparelhamento manual

O outro caminho a ser seguido, no que diz respeito à conexão de aplicações tethering, se reflete num emparelhamento "manual", a partir do apontamento de uma determinada aplicação a ser conectada, presente em uma lista de apps previamente "descobertas" naquele instante. Obviamente, no geral, esse tipo de abordagem provê um controle maior sobre os processos envolvidos, principalmente no quesito conexão, se comparado ao emparelhamento automático. Do ponto de vista técnico, para esta conexão "manual" entre aplicações tethering o desenvolvedor, basicamente, deverá lidar com dois objetos específicos, refletidos em instâncias de duas classes distintas: TTetheringManagerInfo e TTetheringProfileInfo.

Na prática, para sua realização, três requisitos devem ser atendidos. O primeiro, relacionado ao componente Manager

(TTetheringManager) presente no projeto da aplicação, se dá pela chamada ao seu método DiscoverManagers, que tem a incumbência de descobrir outros aplicativos tethering presentes no contexto. Da mesma forma que o método anteriormente citado (AutoConnect), DiscoverManagers também contempla um parâmetro de entrada relacionado ao seu tempo de operação (Timeout), conforme pode ser comprovado em sua assinatura a seguir. Apesar da aparente semelhança, a diferença fundamental entre ambos se dá na questão da finalidade: enquanto AutoConnect irá descobrir e conectar aplicações de forma automática, o método DiscoverManagers se limita a apenas descobrir outras aplicações tethering:

```
procedure DiscoverManagers(Timeout: Cardinal = 1500);
```

Na sequência, o segundo requisito a ser atendido neste cenário, ainda relacionado ao componente Manager, diz respeito à leitura de sua propriedade RemoteManagers, que provê a lista de Managers descobertos. Apenas para reforçar conceitos, vale lembrar que toda aplicação Delphi, seja ela Desktop ou Mobile, que faça uso do recurso de app tethering irá, inevitavelmente, possuir um elemento Manager (TTetheringManager) em sua composição. A seguir, na **Listagem 1**, é mostrada uma exemplificação de leitura desta lista, por meio de uma iteração for-in tradicional. Tecnicamente falando, a partir deste trecho é possível notar que a composição de RemoteManagers se dá por elementos TTetheringManagerInfo que, em código, se reflete em TList<TTetheringManagerInfo>. Em complemento, uma vez identificadas as aplicações tethering, o emparelhamento com alguma delas ocorre pela simples chamada ao método PairManager ainda do elemento Manager, conforme mostra a **Listagem 2**.

Listagem 1. Exemplo de iteração para leitura da lista de Managers remotos

```
1. var
2.  ElementoDescoberto: TTetheringManagerInfo;
3. begin
4.  for ElementoDescoberto in TetheringManager1.RemoteManagers do
5.  begin
6.    // tratativas
7.  end;
8. end;
```

Listagem 2. Exemplo de chamada ao método PairManager

```
1. var
2.  ElementoDescoberto: TTetheringManagerInfo;
3. begin
4.  for ElementoDescoberto in TetheringManager1.RemoteManagers do
5.    TetheringManager1.PairManager(ElementoDescoberto);
6. end;
```

Neste cenário, além do emparelhamento de Managers, para a total interligação entre as aplicações tethering, faz-se necessária também a conexão de Profiles, que caracteriza então o terceiro (e último) requisito exigido nesta abordagem. Para tal, uma atividade semelhante a anterior deve ser realizada, por meio da leitura de outra propriedade do componente TTetheringManager, denominada RemoteProfiles, que mantém a listagem de todos os Profiles remotos detectados. Da mesma forma, tal property se trata de uma Lista, agora de elementos do tipo TTetheringProfileInfo, refletida em código como TList<TTetheringProfileInfo>. Diante disso, a forma natural de leitura desta, ocorre por meio de um processo iterativo, conforme mostrado na exemplificação da **Listagem 3**. Em complemento a isto, ainda sobre o trecho de código exibido, a estabilização de uma conexão a cada Profile encontrado se dá por meio de uma simples chamada ao método Connect do elemento de Profile da aplicação atual.

Listagem 3. Exemplo de iteração para leitura da lista de Profiles remotos

```
1. var
2.  ProfileDescoberto: TTetheringProfileInfo;
3. begin
4.  for ProfileDescoberto in TetheringManager1.RemoteProfiles do
5.    TetheringAppProfile1.Connect(ProfileDescoberto);
6. end;
```

Emparelhamento com autenticação

Independente da abordagem adotada, o emparelhamento entre aplicações tethering pode se dar por meio de uma prévia autenticação, incidindo positivamente sobre as questões de segurança citadas anteriormente. Na prática, este cenário envolve o uso de senha, que deve ser especificada no gerenciador do recurso na aplicação, ou seja, no componente TTetheringManager, conforme mostrado na **Figura 3**. Assim, esta é a informação que deverá ser passada pelas aplicações que venham a se emparelhar com a aplicação atual. Pelo lado técnico, o próprio componente Manager possui um evento, nomeado OnRequestManagerPassword, voltado exclusivamente para essa tratativa.

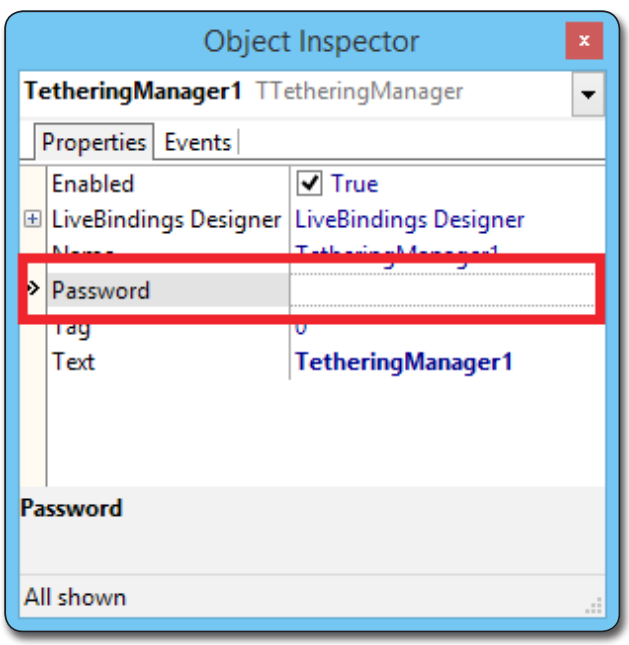


Figura 3. TTetheringManager – propriedade Password

Executando ações compartilhadas

O compartilhamento de ações no cenário Tethering acontece essencialmente pela disponibilização de determinadas ações por parte de uma determinada aplicação, as quais poderão ser executadas remotamente por outras aplicações conectadas. Além disso, as ações citadas aqui nada mais são do que o espelhamento de elementos Actions habituais, presentes em componentes tradicionais como TActionManager e TActionList. Tendo em vista este cenário, para uma aplicação VCL Forms, a presença de um destes elementos seria o suficiente para que eventuais ações fossem definidas, visando o compartilhamento via tethering. A **Figura 4** exibe o formulário, em tempo de Design, pertencente a um simples projeto VCL Forms Application, onde podem ser vistos os componentes necessários para a adição do recurso de app tethering a uma aplicação, conforme já abordados anteriormente, bem como um TActionManager, para as ações.

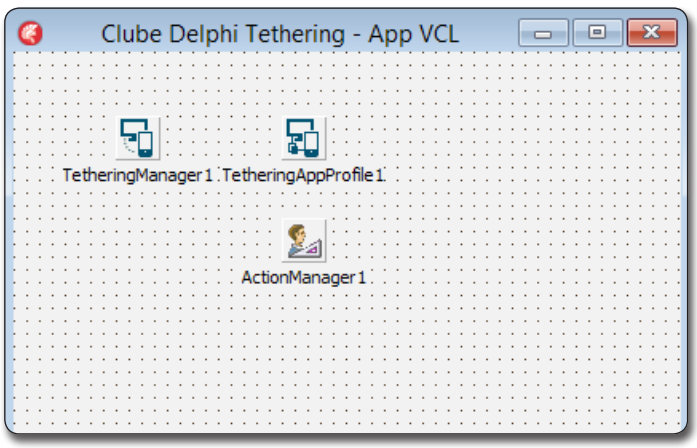


Figura 4. Aplicação VCL de exemplo – componentes

A partir disso, toda a definição de ações se dá no elemento de Profile, que provê uma propriedade voltada exclusivamente a este propósito, com o sugestivo nome de "Actions" (**Figura 5**). Logo, Actions é estruturada como uma coleção de elementos do tipo TLocalAction, em que cada qual se associa a uma Action tradicional. Como forma de ilustrar esta citação com base no exemplo abordado, a **Figura 6** mostra em detalhes algumas ações definidas no ActionManager sendo disponibilizadas automaticamente a um TLocalAction nomeado como "LocalActionClubeDelphi". Em meio as ações listadas está a "ActionMensagemClubeDelphi", definida conforme a seguir.

```
procedure TFrmPrincipal.ActionMensagemClubeDelphiExecute(Sender: TObject);
begin
 ShowMessage('Clube Delphi - Tethering');
end;
```

Isso é o suficiente para que a ação definida possa ser executada por uma aplicação conectada. Sendo assim, mediante o exemplo proposto, o próximo passo natural se dá pela definição de uma segunda aplicação, que terá a incumbência de se conectar a aplicação atual e eventualmente executar a ação compartilhada. Para dar mostras da flexibilidade do recurso, será tomada como base uma nova aplicação, agora do tipo FireMonkey Desktop. Repetindo o mesmo cenário anterior, a **Figura 7** mostra o formulário desta aplicação, já com os componentes necessários envolvidos. A diferença pontual fica por conta do uso de um TActionList, em substituição ao TActionManager utilizado anteriormente, em vista da indisponibilidade do mesmo no contexto FMX.

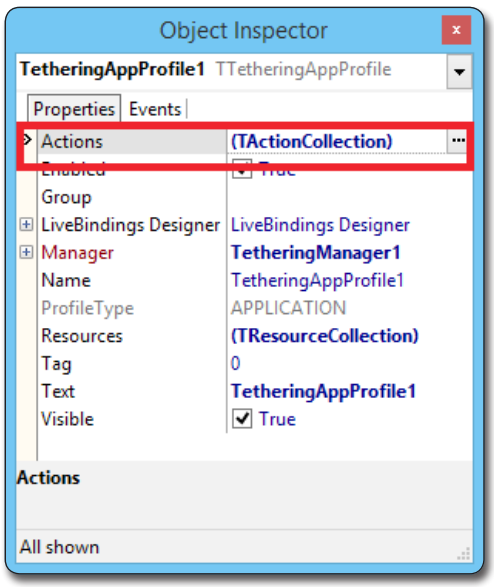


Figura 5. TTetheringAppProfile – propriedade Actions

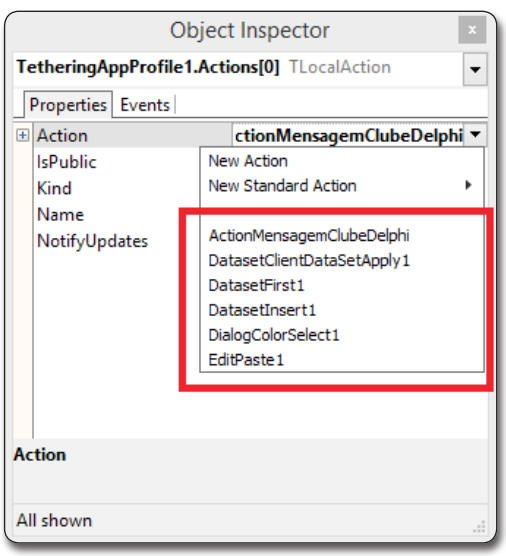


Figura 6. TLocalAction – Ações disponíveis

Em suma, a intenção aqui é associar uma ação local a ação remota definida anteriormente na aplicação VCL. Por este fato, dois requisitos surgem como sendo obrigatórios a esta atividade. O primeiro deles diz respeito ao nome da ação (TLocalAction) a ser criada localmente, que deverá coincidir com o nome da

ação remota, neste caso "LocalActionClubeDelphi". O segundo requisito, não menos importante, refere-se ao tipo (Kind) da ação, que deverá ser Mirror, termo este que numa tradução livre para o português significa "espelho". Essa simples mudança pelo "tipo" da ação faz com que a mesma não seja compartilhada, ao invés disso, durante seu acionamento a ação remota homônima é que será executada. Nota-se então uma referência direta entre ambas (ação local e ação remota). Outro detalhe a ser observado está relacionado ao grupo utilizado, que deverá ser o mesmo em ambas as aplicações. Para este exemplo esta confirmação pode ser observada a partir de um trecho (**Listagem 4**) do código DFM do componente TTetheringAppProfile dos dois projetos envolvidos.

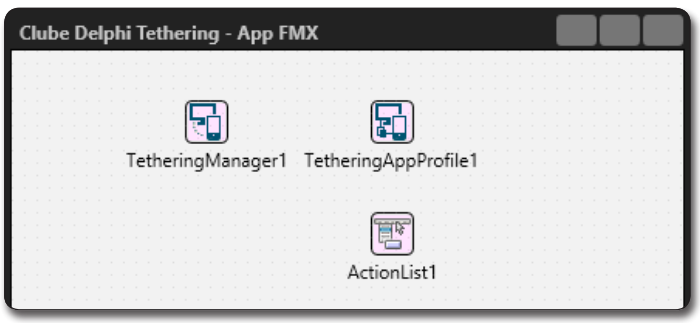


Figura 7. Aplicação FMX de exemplo – componentes

Listagem 4. Configuração do componente TTetheringAppProfile

```
1. //Aplicação VCL
2. object TetheringAppProfile1: TTetheringAppProfile
3.    Group = 'clubedelphi'
4.    ...
5. end
6.
7. //Aplicação FMX
8. object TetheringAppProfile1: TTetheringAppProfile
9.    Group = 'clubedelphi'
10.   ...
11.end
```

Por fim, a plena execução de ambas e o acionamento da ação na aplicação FireMonkey tem como resultado a mensagem mostrada na **Figura 8**, fruto da execução da ação na aplicação VCL. Obviamente, todo este panorama abordado diz respeito a apenas uma, dentre as várias opções de uso deste tipo de funcionalidade.

Consumindo recursos compartilhados

Concluindo esta abordagem essencial do recurso de app tethering no Delphi XE6, conforme já dito anteriormente, além de ações, o mesmo também está apto a compartilhar dados entre as aplicações conectadas. Neste âmbito, os dados envolvidos são tratados como recursos (resources), de forma que um recurso irá envolver um valor. Logo, aplicações tethering são plenamente capazes de ler e também de sobrescrever o valor de um resource. No lado prático, em suma, todo o cenário desta segmentação se dá de forma muito semelhante ao compartilhamento de ações.

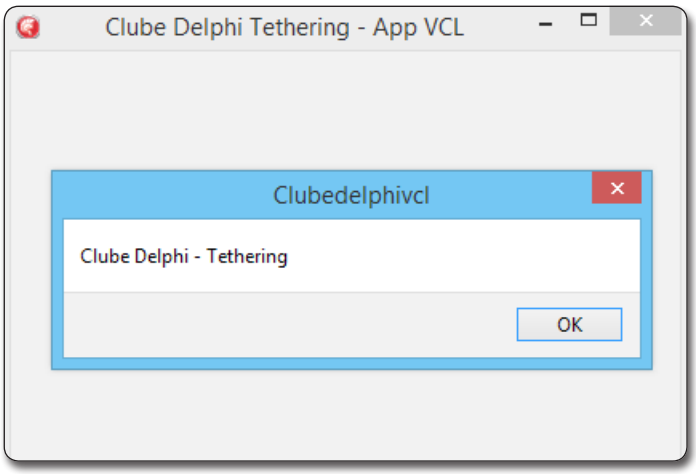


Figura 8. Exemplo em runtime

Traçando um rápido paralelo entre ambas abordagens, ao invés de se definir elementos do tipo TLocalAction, representativos de ações, aqui serão definidos objetos do tipo TLocalResource, na forma de recursos. Nos bastidores, TLocalResource funciona como um wrapper para qualquer tipo de dado padrão do Delphi, tais como String, Double, Integer, Boolean, além de TStream.

Neste cenário, a criação de um recurso compartilhado, assim como ocorre para ações, se dá no componente de Profile (TTetheringAppProfile), mais especificamente em sua propriedade Resources (**Figura 9**), que se mostra como uma coleção de elementos TLocalResource. A seguir é exposto um trecho do código DFM do componente (**Listagem 5**), mediante a definição de um novo Resource denominado "ResourceAppVCL", bem como sua representação, vista na **Figura 10**.

Listagem 5. Configuração do componente TTetheringAppProfile

```
1. object TetheringAppProfile1: TTetheringAppProfile
2.    Manager = TetheringManager1
3.    Group = 'clubedelphi'
4.    Resources = <
5.      item
6.        Name = 'ResourceAppVCL'
7.        IsPublic = True
8.      end>
9.    ...
10.end
```

Apesar da definição do recurso poder ser feita de forma totalmente visual, ainda em tempo de Design, a atribuição de seu valor ocorre via código, em runtime, conforme pode ser visto logo adiante. Neste esboço, foi lhe atribuído um simples valor textual (string) com os dizeres "Resource from AppVCL". Isto é o suficiente para que o mesmo esteja apto a ser "consumido" por outras aplicações conectadas, como é o caso da aplicação FireMonkey neste exemplo:

```
TetheringAppProfile1.Resources.FindByName('ResourceAppVCL').Value :=
  'Resource from AppVCL';
```

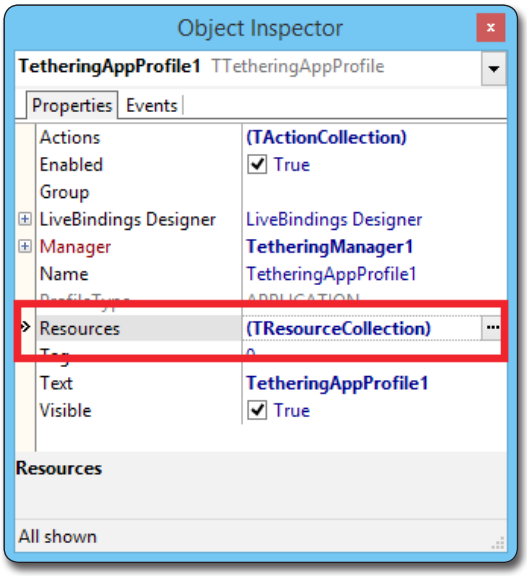


Figura 9. TTetheringAppProfile – propriedade Resources

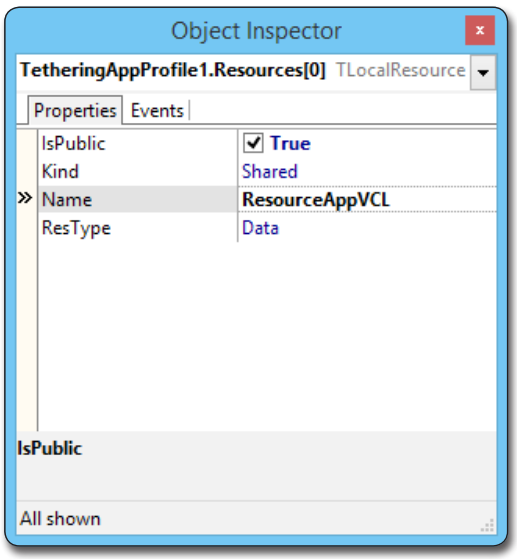


Figura 10. Definição de um recurso no componente de Profile

Seguindo por esta linha, a consequente chamada ao recurso compartilhado, já na aplicação FMX, poderia ser feita da seguinte forma mostrada na **Listagem 6**.

Aqui dois pontos merecem ser destacados. O primeiro diz respeito ao método GetRemoteResourceValue do componente TTetheringAppProfile, que tem a incumbência de realizar o request de um recurso remoto compartilhado. Para isso, ele faz uso de dois parâmetros, os quais se refletem em um elemento do tipo TTetheringProfileInfo, disponível na lista de Profiles descobertos (propriedade RemoteProfiles do elemento Manager), bem como o nome do recurso a ser solicitado (neste caso "ResourceAppVCL"). Logo, o retorno do método é uma instância de um objeto TRemoteResource, cujo valor é trazido à tona por meio de sua propriedade Value. De posse disso, uma chamada a AsString faz com que o valor possa ser exibido em tela através de um simples ShowMessage. Por conseguinte, já em runtime, toda esta chamada resulta na mensagem mostrada na **Figura 11**.

Listagem 6. Configuração do componente TTetheringAppProfile

```
1. var
2.  RemoteProfileInfo: TTetheringProfileInfo;
3.  ValorRecurso: string;
4. begin
5.  ValorRecurso := TetheringAppProfile1.GetRemoteResourceValue(
6.   TetheringManager1.RemoteProfiles[0],
7.   'ResourceAppVCL').Value.AsString;
8.  ShowMessage(ValorRecurso);
9. end;
```

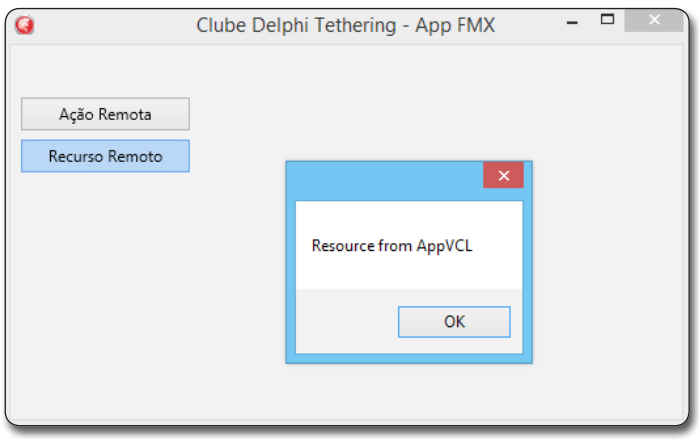


Figura 11. Aplicação FMX – chamada a um recurso remoto

Indo além, para casos reais onde há a presença de vários recursos envolvidos, o componente de Profile fornece outro método direcionado, denominado GetRemoteProfileResources, que retorna uma lista dos recursos remotos compartilhados, provenientes de determinada aplicação. A indicação desta é feita pela passagem de parâmetro de um elemento TTetheringProfileInfo, tal como se deu com o método GetRemoteResourceValue. Novamente de forma ilustrativa, a **Listagem 7** exibe um exemplo deste tipo de atividade.

Listagem 7. Acessando recursos remotos com GetRemoteProfileResources

```
1. var
2.  RemoteProfileInfo: TTetheringProfileInfo;
3.  RemoteResource: TRemoteResource;
4. begin
5.  for RemoteProfileInfo in TetheringManager1.RemoteProfiles do
6.  begin
7.   for RemoteResource in TetheringAppProfile1.
8.     GetRemoteProfileResources(RemoteProfileInfo) do
9.     //Acesso às propriedades do elemento RemoteResource
10.  end;
11.end;
```

Mediante a codificação apresentada, o passo inicial se deu pela leitura da propriedade RemoteProfiles do componente TTetheringManager como forma de acesso a todos os Profiles remotos localizados. Conforme já mencionado, no código, cada Profile descoberto se reflete em um objeto do tipo TTetheringProfileInfo. Assim, a seguir, para cada elemento deste, é feita uma chamada ao método GetRemoteProfileResources do componente de Profile, a fim de se ter acesso a cada um de seus eventuais recursos compartilhados.

Trabalhando com recursos temporários

O recurso de tethering provê ainda uma alternativa ao uso de recursos compartilhados que supre a necessidade da definição efetiva de um resource, tal como mostrado anteriormente, caracterizando então um recurso temporário. Assim, ao invés da localização de um recurso definido, é utilizada uma simples chamada a um determinado método Send que irá enviar dados momentâneos às aplicações conectadas. Na prática, dois são os métodos disponíveis com este intuito – SendString e SendStream – cujas assinaturas são mostradas a seguir:

```
procedure SendString(const AProfile: TTetheringProfileInfo; const Description, AString: string);

function SendStream(const AProfile: TTetheringProfileInfo; const Description: string; const AStream: TStream): Boolean;
```

Mediante os parâmetros envolvidos, AProfile refere-se ao Profile Remoto que irá receber a informação enviada. Description, conforme sua própria denominação sugere, funciona como um identificador descritivo do conteúdo, enquanto que AStream/AString referem-se ao conteúdo a seguir:

```
procedure SendString(const AProfile: TTetheringProfileInfo; const Description, AString: string);

function SendStream(const AProfile: TTetheringProfileInfo; const Description: string; const AStream: TStream): Boolean;
```

O tethering pode trazer novos desafios ao desenvolvimento, o primeiro se dá pelo próprio aprofundamento do assunto, centrado no estudo da dupla de componentes envolvida. O segundo pode ocorrer pela interatividade prática com os diversos tipos de aplicações nativas providas atualmente pela ferramenta, por exemplo, a comunicação entre aplicações FMX, Desktop e Mobile simultaneamente.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE6 – Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE6 – Download Trial
https://downloads.embarcadero.com/free/rad_studio

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Behavioral Design Patterns em Delphi

Controle o comportamento de suas classes

Este artigo é do tipo mentoring

saiba mais: www.devmedia.com.br/mentoring-saibamais

Cenário

Este artigo aborda os padrões de projeto do tipo comportamentais (behavioral). Em especial, os padrões de projeto da categoria comportamentais são extremamente úteis para criar softwares mais fáceis de serem mantidos, evoluídos, promovendo reutilização de código, uso de boas práticas de programação orientada a objetos, redução de custos com manutenção, já que permitem que alterações em um determinado sistema de software não quebrem outras funcionalidades, reduzindo a dependência entre classes, através do uso intensivo de abstrações. Enquanto os padrões criacionais resolvem problemas relacionados à criação de objetos, e os estruturais aos modelos de estruturas de classes, os padrões comportamentais resolvem problemas relacionados principalmente à interação e comunicação entre objetos, que normalmente causam forte acoplamento e dependência. Para cada padrão de projeto, serão apresentados as suas intenções, benefícios, vantagens e participantes conforme definidos no livro do GoF. A seguir, será apresentado um exemplo prático em Delphi mostrando como criar um framework com as classes de cada padrão e uma aplicação que consome este framework. O diagrama UML de classes é apresentado para mostrar o relacionamento entre os participantes.

A intenção do padrão Observer é definir uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. O padrão pode ser aplicado quando: uma abstração tem dois aspectos, um dependente do outro. Encapsulando esses aspectos em objetos separados, permite-se variá-los e reutilizá-los independentemente; quando uma mudança em um objeto exige mudanças em outros, e você não sabe quantos necessitam ser mudados; quando um objeto deveria ser capaz de notificar outros objetos sem fazer hipóteses, ou usar informações, sobre quem são esses objetos. Em outras palavras, você não quer que esses objetos sejam fortemente acoplados. Entre os principais benefícios podemos citar: acoplamento abstrato entre Subject e Observer e suporte para comunicação do tipo "broadcast".

Participantes:

• Subject – conhece os seus observadores. Um número qualquer de objetos Observer pode observar um Subject;

• Observer – define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um Subject;

• Concrete Subject – armazena estados de interesse para objetos ConcreteObserver; envia uma notificação para os seus observadores quando seu estado muda;

• ConcreteObserver – mantém uma referência para um objeto ConcreteSubject; armazena estados que deveriam permanecer consistentes com os do Subject; implementa a interface de atualização de Observer, para manter seu estado consistente como o do Subject;

No Delphi, a classe TComponent implementa o padrão Composite, permitindo que um objeto seja formado a partir da composição de vários outros componentes, guardando internamente uma coleção que referencia cada uma destes, chamada FComponents, do tipo TList<TComponent> (coleção genérica de componentes). Isto permite, por exemplo, que o Delphi implemente o mecanismo de componente "proprietário", mais conhecido como Owner, que gerencia o ciclo de vida de um objeto conforme a composição. TComponent define ainda uma lista de observadores, armazenados no atributo FObservers, do tipo TObservers (você pode verificar esta estrutura na unit System.Classes). Podemos notar uma implementação mais antiga do padrão Observer em TComponent, através do mecanismo de Notification (notificação). O método Notification de TComponent recebe como parâmetro

um componente e uma operação, do tipo TOperation (opInsert, opRemove). FreeNotication é um método que faz uso do padrão Observer para monitorar quando um componente é excluído, por exemplo, do form designer. É graças a esse mecanismo de observação que, por exemplo, quando excluímos uma instância de algum TDataSet como um ClientDataSet que esteja sendo referenciado na propriedade DataSet de um TDataSource, o mesmo receberá uma notificação informando que houve uma exclusão, removendo assim a referência ao objeto e evitando um Access Violation.

A **Listagem 1, Listagem 2** e **Figura 1** mostram um exemplo de implementação do padrão Observer em Delphi.

Listagem 1. Observer

```
unit uFramework;

interface

uses
 Generics.Collections;

type
 // Observer
 Observador = class abstract
  public procedure Update(); virtual; abstract;
 end;

 // Subject
 Observado = class abstract
  private _observadores: TList<Observador>;
  public procedure Attach(obs: Observador);
  public constructor Create();
  public destructor Destroy(); override;
  public procedure Notity();
 end;

 // Concrete Subject
 Balanco = class (Observado)
  // state é true se balanço está em andamento
  private _state: Boolean;
  public property State: boolean read _state write _state;
  public constructor Create();
  public procedure Iniciar();
  public procedure Finalizar();
 end;

 // Concrete Observer
 Venda = class (Observador)
  private _state: boolean;
  private _balanco: Balanco;
  public procedure Iniciar();
  public procedure Update(); override;
  public constructor Create(b: Balanco);
 end;

implementation

{ Subject }

procedure Observado.Attach(obs: Observador);
begin
 Self._observadores.Add(obs);
end;

constructor Observado.Create();
begin
 Self._observadores := TList<Observador>.Create();
end;

destructor Observado.Destroy();
begin
 Self._observadores.Free();
 inherited;
end;

procedure Observado.Notity();
var
 obs: Observador;
begin
 // broadcast (avisar todos os observadores)
 for obs in _observadores do
  obs.Update();
end;

{ Balanco }

constructor Balanco.Create();
begin
 inherited Create();
 Self._state := false;
end;

procedure Balanco.Finalizar();
begin
 _state := false;
 WriteLn('Balanço finalizado...');
 // notifica os observadores
 Notity();
end;

procedure Balanco.Iniciar();
begin
 _state := true;
 WriteLn('Balanço iniciado...');
 // notifica os observadores
 Notity();
end;

{ Venda }

constructor Venda.Create(b: Balanco);
begin
 self._balanco := b;
end;

procedure Venda.Iniciar();
begin
 // state = true se balanço foi iniciado
 if (not _state) then
  Writeln('Venda iniciada')
 else
  Writeln('Não pode iniciar venda pois balanço está em andamento');
end;

procedure Venda.Update();
begin
 _state := _balanco.State;
end;

end.
```

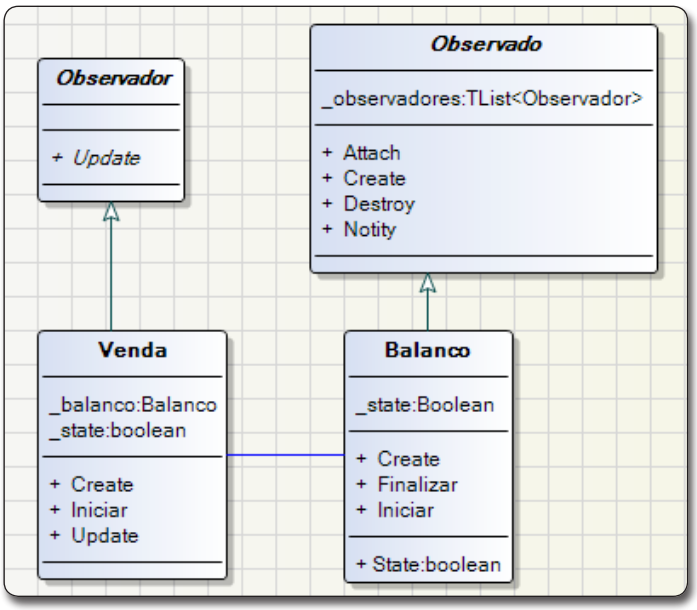


Figura 1. Estrutura de classes do exemplo usando Observer

Listagem 2. Usando um Observer

```
program Observer;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  b: Balanco;
  v: Venda;
begin
  // Concrete Subject
  b := Balanco.Create();
  // Concrete Observer
  v := Venda.Create(b);
  // adicionar os observadores
  b.Attach(v);
  // processo...
  b.Iniciar();
  b.Finalizar(); // comentar essa linha para testar
  // pode vender?
  v.Iniciar();
  Readln;
end.
```

Aqui foram definidas duas classes, chamadas de Venda e Balanco, representando operações em um sistema de vendas. Neste caso, quando um balanço está em andamento, não é possível iniciar uma venda. Dessa forma, uma classe precisa monitorar a outra, verificando constantemente o seu estado. Normalmente programadores fazem uso do componente TTimer para tarefas deste tipo, definindo um tempo curto para disparo do evento de verificação (ex. 1 segundo), e constantemente consultando propriedades ou estado de componentes públicos de um formulário alvo para atualizar o formulário de origem. Esta, além de não ser uma boa prática devido ao forte acoplamento causado, cria um overhead desnecessário de notificações, visto que a verificação de estado precisa ser feita a cada segundo. No exemplo apresentado, a classe Balanco é nossa classe observada, enquanto Venda é a classe observadora. Quando um balanço é iniciado (método Iniciar de Balanco), o método Notify é invocado, que por sua vez varre toda a coleção de observadores avisando que seu estado foi modificado. Veja que na abordagem com Timer, esse processo seria o inverso. Aqui quem sofre o evento, e somente neste momento, é quem vai notificar as demais classes observadoras, evitando várias chamadas constantes para monitoramento de estado. Quando uma venda é iniciada, esta verifica se não há um balanço em andamento, desta forma, não permitindo que a operação seja executada.

State

A intenção do padrão State é permitir que um objeto altere seu comportamento quando seu estado interno muda.

O padrão pode ser aplicado: quando o comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado; operações têm comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto. Frequentemente, várias operações conterão essa mesma estrutura condicional. O padrão State coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, que pode variar independentemente de outros objetos. Entre alguns benefícios podemos citar: o padrão confina comportamento específico de estados e particiona o comportamento para estados diferentes; ele torna explícitas as transações de estado; objetos state podem ser compartilhados (Gamma et al., 1995).

Participantes:

- Context – define a interface de interesse para os clientes; mantém uma instância de uma subclasse ConcreteState que define o estado corrente;
- State – define uma interface para o encapsulamento associado com um determinado estado do Context;
- ConcreteState Subclasses – cada subclasse implementa um comportamento associado com um estado do Context.

A **Listagem 3, Listagem 4** e **Figura 2** mostram um exemplo de implementação do padrão State em Delphi. Aqui temos um pequeno framework que representa classes de conexão a banco de dados. De acordo com o estado da conexão, o comportamento do objeto muda. A classe base abstrata chamada de ConnectionState define dois métodos, chamados Open e Close, que recebem como parâmetro uma classe de conexão. Existem duas classes derivadas (concretas) de ConnectionState, que representam os possíveis estados: ConnectionClosed e ConnectionOpened. Estas classes através de polimorfismo sobrescrevem os métodos Open e Close de ConnectionState, configurando o atributo state que representa o estado da conexão (classe Connection).

Listagem 3. State

```
unit uFramework;

interface

type

 Connection = class; // forward

 // State
 ConnectionState = class abstract
  public procedure Open(context: Connection); virtual; abstract;
  public procedure Close(context: Connection); virtual; abstract;
 end;

 // Concrete State
 ConnectionClosed = class(ConnectionState)
  public procedure Open(context: Connection); override;
  public procedure Close(context: Connection); override;
 end;

 // Concrete State
 ConnectionOpened = class(ConnectionState)
  public procedure Open(context: Connection); override;
  public procedure Close(context: Connection); override;
 end;

 // Context
 Connection = class
  private _state: ConnectionState;
  function getConnectionState(): ConnectionState;
  procedure setConnectionState(const Value: ConnectionState);
  public constructor Create(state: ConnectionState);
  public property State: ConnectionState
   read getConnectionState write setConnectionState;
  public procedure Open();
  public procedure Close();
 end;

implementation

{ Connection }

procedure Connection.Close();
begin
 self._state.Close(self);
end;

constructor Connection.Create(state: ConnectionState);
begin
 self._state := state;
end;

function Connection.getConnectionState(): ConnectionState;
begin
 result := Self._state;
end;

procedure Connection.Open();
begin
 self._state.Open(self);
end;

procedure Connection.setConnectionState(
 const Value: ConnectionState);
begin
 self._state := Value;
 WriteLn('State: ' + _state.ClassType.ClassName);
end;

{ ConnectionClosed }

procedure ConnectionClosed.Close(context: Connection);
begin
 context.State := ConnectionClosed.Create();
end;

procedure ConnectionClosed.Open(context: Connection);
begin
 context.State := ConnectionOpened.Create();
end;

{ ConnectionOpened }

procedure ConnectionOpened.Close(context: Connection);
begin
 context.State := ConnectionClosed.Create();
end;

procedure ConnectionOpened.Open(context: Connection);
begin
 context.State := ConnectionOpened.Create();
end;

end.
```

Em outras palavras, o state (estado) de uma classe Connection pode ser representado por dois tipos que também são classes (ConnectionClosed e ConnectionOpened), permitindo assim variar o comportamento conforme este estado.

Strategy

Este padrão tem por objetivo definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizem. Esse padrão pode ser aplicado quando muitas classes têm propósitos semelhantes e divergem apenas no seu comportamento (implementação). Ou ainda, quando é necessário encapsular a solução para um problema resolvido por um ou mais algoritmos, dos quais classes clientes não devem tomar conhecimento.

Listagem 4. Usando um State

```
program State;

{$APPTYPE CONSOLE}
{$R *.res}

uses
 System.SysUtils,
 uFramework in 'uFramework.pas';
var
 con: Connection;
begin
 con := Connection.Create(ConnectionOpened.Create());
 // mudar o estado do objeto
 con.Open();
 con.Close();
 ReadLn;
end.
```

Este padrão traz alguns benefícios, como a eliminação de comandos condicionais para a seleção de um comportamento desejado (algoritmo) para resolver um problema. Ao usar herança, Strategy encapsula esses comportamentos em subclasses e usa polimorfismo (que é dinâmico) para substituir as condicionais estáticas, abrindo inclusive a arquitetura para fácil inclusão de novas estratégias para solucionar o mesmo problema (Gamma et al., 1995).

Participantes:

• Strategy – define uma interface comum para todos os algoritmos suportados. Context usa esta interface para chamar o algoritmo definido por uma ConcreteStrategy;

• ConcreteStrategy – implementa o algoritmo usando a interface Strategy;

• Context – mantém uma referência para um objeto Strategy; pode definir uma interface que permite a Strategy acessar seus dados;

O padrão Strategy resolve um dos piores mau cheiros no código, mais conhecidos como code smells ou bad smells, relacionados ao uso excessivo de instruções condicionais. Existem diversas refatorações destinadas à simplificação de expressões condicionais. Muitas vezes encontramos essa lógica em algoritmos, porque eles tendem a crescer, tornando-se mais sofisticado ao longo do tempo. O padrão Strategy ajuda a gerenciar a complexidade no momento em que se tem vários algoritmos destinados a resolver um determinado problema. Testar várias possibilidades com sentenças case ou if/elses aninhados tende a duplicar código. Se um novo teste for necessário na sentença, para comportar uma nova possibilidade, e esse teste for feito em vários locais, então várias alterações serão necessárias. Muitas vezes os testes podem ser substituídos por polimorfismo, que é uma das principais técnicas de programação orientada a objetos, usada internamente pelo padrão Strategy. A inclusão de muitos testes condicionais aumenta gradativamente a complexidade do código conforme o mesmo vai evoluindo ao longo do tempo, tornando sua manutenção ainda mais complexa. Quando são realizados testes unitários em código, analisar a cobertura de código afetada por testes se torna um processo trabalhoso, pois é necessário incluir lógica para testar cada uma das sentenças condicionais. E finalmente, testes condicionais dificultam a leitura e interpretação por parte de desenvolvedores que necessitam realizar manutenções corretivas no sistema de software.

Vamos a um exemplo no próprio Delphi. A classe TDataSource possui uma referência a um TDataSet, a qual existe desde o Delphi 1, portanto, adaptável a mudanças (estamos no XE6 e ela ainda existe). TDataSet é uma abstração, que possui dezenas de implementações concretas. Quando você chama o método Post, por exemplo, de qualquer tipo de TDataSet, está na verdade chamando um método virtual polimórfico. Normalmente, quando construímos formulários base de cadastro em Delphi, colocamos apenas um TDataSource em tela, mantendo TDataSets de acesso a dados separados em um TDataModule. Estes TDataSets são implementações concretas da abstração, como TClientDataSet (DataSnap), TADOQuery (ADO) ou TFDQuery (FireDac). Agora imagine que neste formulário base você precisa invocar o método Post para gravar os dados do formulário, de forma a tornar a implementação aberta independente de qual engine de acesso a dados utiliza. Facilmente podemos fazer uma chamada como DataSource1.DataSet.Post(). Ou seja, chamamos o método Post do DataSet que está sendo atualmente referenciado pelo TDataSource do formulário base, não importa qual tipo de TDatSet seja (TClientDataSet, TFBQuery etc). Em tempo de execução, o método Post da correta classe concreta será invocado, através de polimorfismo, sem a necessidade de testes condicionais. Veja que neste caso, TDataSource "esconde" a estratégia de gravação usada para persistir dados de Data Controls a ele vinculados, algo muito semelhante ao que sugere o padrão Strategy. De fato, se feita uma boa arquitetura, é perfeitamente possível substituir todo um engine de acesso a dados mantendo formulários intactos. Você troca de um TADOQuery configurado para persistir dados em um banco SQL Server por um TFDQuery, dessa forma, fazendo o intercâmbio dos "algoritmos", atingindo o mesmo resultado esperado (gravar no SQL Server) usando uma outra estratégia.

A **Listagem 5, Listagem 6** e **Figura 3** mostram um exemplo de implementação do padrão Strategy em Delphi. Aqui definimos um framework de autenticação de usuários, com uma classe abstrata que define um método chamado Autenticar, recebendo usuário e senha. A ideia é que dado um sistema, possamos trocar a forma como um usuário é autenticado, atingindo o mesmo objetivo (validar as credenciais). Ou seja, permitir intercambiar diferentes formas de autenticação, dependendo do sistema. No exemplo, três formas de autenticação são implementadas: via servidor SQL, arquivos XML ou Web Services (para fins didáticos, estaremos implementando o teste condicional de forma estática no

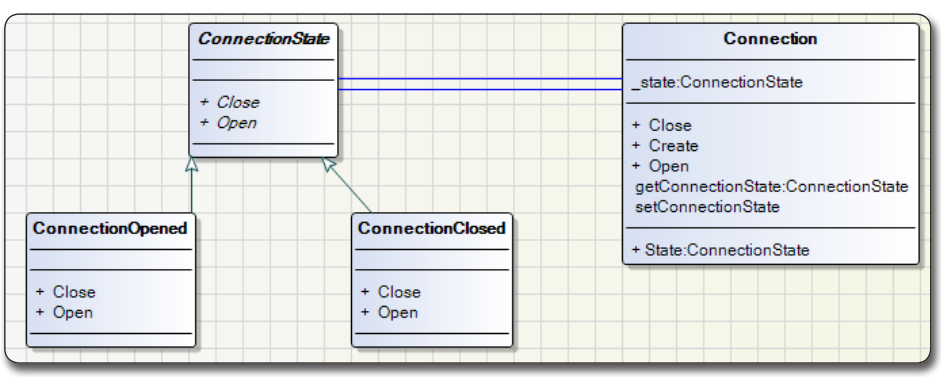


Figura 2. Estrutura de classes do exemplo usando State

Listagem 5. Strategy

```
unit uFramework;

interface

type

 // Strategy
 Autenticacao = class abstract
   public function Autenticar(UserName, Password: string): boolean;
   virtual; abstract;
 end;

 // Context (Open Closed Principle - OCP)
 Context = class
   private _autenticacao: Autenticacao;
   public constructor Create(AAutenticacao: Autenticacao);
   public destructor Destroy(); override;
   public function Autenticar(Usuario, Senha: string): boolean;
 end;

 // Concrete Strategy
 AutenticacaoSQL = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean; override;
 end;

 // Concrete Strategy
 AutenticacaoWebService = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean; override;
 end;

 // Concrete Strategy
 AutenticacaoXML = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean;
 end;


implementation

{ AutenticacaoSQL }

function AutenticacaoSQL.Autenticar(Usuario, Senha: string): boolean;
begin
  // valida usuário usando conexão com SQL Server
  result := (Usuario = 'guinther') and (Senha = '123');
end;

{ AutenticacaoWebService }

function AutenticacaoWebService.Autenticar(Usuario, Senha: string): boolean;
begin
  // valida usuário usando Web Service
  result := (Usuario = 'guinther') and (Senha = '123');
end;

{ AutenticacaoXML }

function AutenticacaoXML.Autenticar(Usuario, Senha: string): boolean;
begin
  // valida usuário usando dados de um arquivo XML
  result := (Usuario = 'guinther') and (Senha = '123');
end;

{ Context }

constructor Context.Create(AAutenticacao: Autenticacao);
begin
  Self._autenticacao := AAutenticacao;
end;

destructor Context.Destroy();
begin
  Self._autenticacao.Free();
  inherited;
end;

function Context.Autenticar(Usuario, Senha: string): boolean;
begin
  result := _autenticacao.Autenticar(Usuario,Senha);
end;

end.
```

código, e não usando a tecnologia propriamente dita). A aplicação cliente pode então, através de Context, escolher qualquer uma destas formas de autenticação, sem que para isso sejam necessários testes condicionais. Tudo é resolvido em tempo de execução através de polimorfismo.

Template Method

Este padrão tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo. Este padrão pode ser aplicado quando existe um algoritmo que resolve um problema a partir de uma série de operações (partes), chamadas da mesma forma, porém estas formas podem variar. Esta variação é então explicitada em subclasses, usando polimorfismo. Um método na classe base nesse caso funciona como um "template", que chama operações em pontos específicos do código, de forma apropriada. Subclasses então apenas se encarregam de dar uma forma concreta a estes pontos já chamados.

Listagem 6. Usando Strategy

```
program Strategy;

{$APPTYPE CONSOLE}
{$R *.res}
uses
  System.SysUtils,
  uFramework in 'uFramework.pas';
var
  ctx: Context;
  Usuario, Senha: string;
begin
  Writeln('Usuario:');
  Readln(Usuario);
  Writeln('Senha:');
  Readln(Senha);
  // trocar aqui forma de autenticação para teste
  ctx := Context.Create(AutenticacaoWebService.Create());
  if ctx.Autenticar(Usuario,Senha) then
   Writeln('Usuário autenticado com sucesso')
  else
   Writeln('Credenciais inválidas');
  Readln;
end.
```

É uma espécie de inversão de controle, pois nesse caso é a classe base que chama indiretamente métodos das classes filhas, e não o contrário. Um benefício do uso do padrão é evitar redundância de código, nesse caso, a chamada a vários métodos em sequência.

Participantes:

• AbstractClass – define as operações primitivas abstratas que as subclasses concretas definem para implementar passos de um algoritmo; implementa um método-template que define o esqueleto de um algoritmo;

• ConcreteClass – implementa as operações primitivas para executar os passos específicos invariantes do algoritmo;

No Delphi, quando chamamos o método Post de qualquer tipo de TDataSet, ele internamente chama outros métodos polimórficos em sequência, como DataEvent, DoBeforePost, Resync e DoAfterPost. Em especial, DoBeforePost e DoAfterPost são métodos de despacho que verificam se você por acaso definiu um manipulador (handler) para os eventos BeforePost e AfterPost. Se por acaso você estiver criando um novo engine de acesso a dados, não pode usar os eventos para efetuar uma operação antes ou depois da gravação, pois os eventos são usados pelo implementador do sistema no DataModule, não pelo projetista do componente. Nesse caso, é necessário dar um override nos métodos polimórficos. Porém, note que você simplesmente implementa os métodos BeforePost e AfterPost, mas não precisa chamá-los, pois isso já é feito por Post em TDataSet. A ordem de chamada já está definida, você só precisa dizer o como. Before é antes e After é depois, isso é imutável e por isso está implementado na forma de um padrão Template Method, sem replicação.

A **Listagem 7, Listagem 8** e **Figura 4** mostram um exemplo de implementação do padrão Template Method em Delphi.

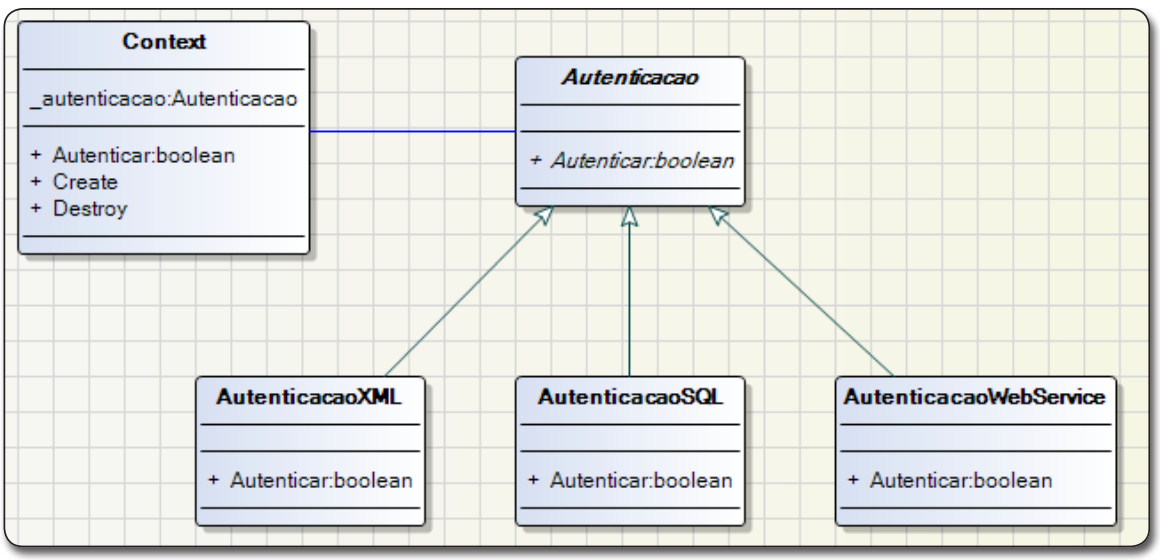


Figura 3. Estrutura de classes do exemplo usando Strategy

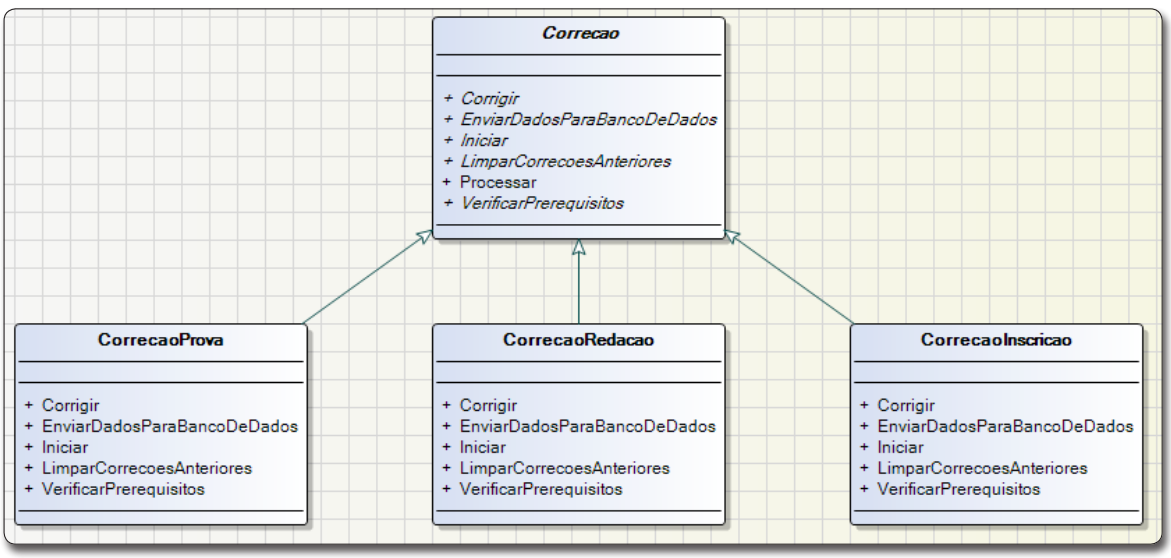


Figura 4. Estrutura de classes do exemplo usando Template Method

Listagem 7. Template Method

```
unit uFramework;

interface

type
 // Abstract Class
 Correcao = class abstract
  public procedure Corrigir(); virtual; abstract;
  public procedure VerificarPrerequisitos(); virtual; abstract;
  public procedure EnviarDadosParaBancoDeDados(); virtual; abstract;
  public procedure LimparCorrecoesAnteriores(); virtual; abstract;
  public procedure Iniciar(); virtual; abstract;
  // Template Method
  public procedure Processar();
 end;

 CorrecaoProva = class(Correcao)
  public procedure Corrigir(); override;
  public procedure VerificarPrerequisitos(); override;
  public procedure EnviarDadosParaBancoDeDados(); override;
  public procedure LimparCorrecoesAnteriores(); override;
  public procedure Iniciar(); override;
 end;

 CorrecaoRedacao = class(Correcao)
  public procedure Corrigir(); override;
  public procedure VerificarPrerequisitos(); override;
  public procedure EnviarDadosParaBancoDeDados(); override;
  public procedure LimparCorrecoesAnteriores(); override;
  public procedure Iniciar(); override;
 end;

 CorrecaoInscricao = class(Correcao)
  public procedure Corrigir(); override;
  public procedure VerificarPrerequisitos(); override;
  public procedure EnviarDadosParaBancoDeDados(); override;
  public procedure LimparCorrecoesAnteriores(); override;
  public procedure Iniciar(); override;
 end;

implementation

{ Correcao }

// este é o template method
procedure Correcao.Processar();
begin
 // orquesta a chama de métodos virtuais
 Iniciar();
 VerificarPrerequisitos();
 LimparCorrecoesAnteriores();
 Corrigir();
 EnviarDadosParaBancoDeDados();
end;
```


CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs

SQL magazine

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

http://www.devmedia.com.br/cursos/banco-de-dados

(21) 3382-5038

DEVMEDIA

Continuação: Listagem 7. Template Method

```
{ CorrecaoProva }

procedure CorrecaoProva.Corrigir();
begin
 Writeln('Corrigindo prova...');
end;

procedure CorrecaoProva.EnviarDadosParaBancoDeDados();
begin
 Writeln('Enviando dados da prova para o BD...');
end;

procedure CorrecaoProva.Iniciar();
begin
 Writeln('Iniciando processo de correção da prova...');
end;

procedure CorrecaoProva.LimparCorrecoesAnteriores();
begin
 Writeln('Limpando dados de correções anteriores de provas...');
end;

procedure CorrecaoProva.VerificarPrerequisitos();
begin
 Writeln('Verificando pré-requisitos para correção da prova...');
end;

{ CorrecaoRedacao }

procedure CorrecaoRedacao.Corrigir();
begin
 Writeln('Corrigindo redação pela nota...');
end;

procedure CorrecaoRedacao.EnviarDadosParaBancoDeDados();
begin
 Writeln('Enviando dados da redação para o BD...');
end;

procedure CorrecaoRedacao.Iniciar();
begin
 Writeln('Iniciando processo de correção da redação...');
end;

procedure CorrecaoRedacao.LimparCorrecoesAnteriores();
begin
 Writeln('Limpando dados de correções anteriores de redação...');
end;

procedure CorrecaoRedacao.VerificarPrerequisitos();
begin
 Writeln('Verificando pré-requisitos para correção da redação...');
end;

{ CorrecaoInscricao }

procedure CorrecaoInscricao.Corrigir();
begin
 Writeln('Processando inscrição...');
end;

procedure CorrecaoInscricao.EnviarDadosParaBancoDeDados();
begin
 Writeln('Enviando dados da inscrição para o BD...');
end;

procedure CorrecaoInscricao.Iniciar();
begin
 Writeln('Iniciando processo de inscrição...');
end;

procedure CorrecaoInscricao.LimparCorrecoesAnteriores();
begin
 Writeln('Limpando dados de inscrições anteriores...');
end;

procedure CorrecaoInscricao.VerificarPrerequisitos();
begin
 Writeln('Verificando pré-requisitos para inscrição...');
end;

end.
```

É apresentado um pequeno framework para correções de provas em sistemas de vestibular / concursos. Em sistemas desse tipo, temos três tipos de processamento: leitura da ficha de inscrição, correção do cartão de respostas da prova e leitura da nota dada pelo avaliador para a redação. Nestes três tipos de processamento (que foram chamados de Correcao, classe base abstrata), temos vários métodos de mesmo nome que devem ser chamados na mesma sequência, porém, têm implementação diferente de acordo com o tipo de leitura. Por exemplo, qualquer tipo de correção deve iniciar pela verificação de pré-requisitos (para fazer a prova o candidato deve estar inscrito), correção das repostas propriamente ditas (processamento do gabarito para uma prova ou verificação da nota para uma redação), envio das informações processadas para o banco de dados e assim por diante. Neste caso, um único método é definido na classe base, chamado Processar. Ele invoca a sequência correta de métodos, neste caso, Iniciar, VerificarPrerequisitos, LimparCorrecoes-Anteriores, Corrigir e EnviarDadosParaBancoDeDados. Ou seja, ele é o Template Method. As classes descendentes então simplesmente indicam como esses métodos são implementados, através de polimorfismo, mas não a

Listagem 8. Usando Template Method

```
program TemplateMethod;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  // Programe para uma interface/abstração
  prova: Correcao;
  redacao: Correcao;
  inscricao: Correcao;
begin
  inscricao := CorrecaoInscricao.Create();
  inscricao.Processar();
  prova := CorrecaoProva.Create();
  prova.Processar();
  redacao := CorrecaoRedacao.Create();
  redacao.Processar();
  Readln;
end.
```

ordem na qual são chamados. É uma espécie de "polimorfismo" ao contrário, pois a classe base é quem define como os métodos são chamados. Observe que isso reduz a duplicação de código, caso contrário, seria necessário estabelecer essa ordem de chamada em todas as classes de correção.

Neste artigo foram apresentados os padrões GoF, da categoria Comportamentais. Vimos como o próprio Delphi faz uso desses padrões em sua estrutura de classes, como na VCL, IDE ou compilador.

Foram apresentados exemplos práticos em Delphi, comprovando como o uso destes padrões ajuda a tornar sistemas de software mais fáceis de serem mantidos e evoluídos, evitando duplicação de código, diminuindo o acoplamento entre classes e promovendo a modularização. Vimos que a maioria dos padrões está fundamentada sobre as principais técnicas da orientação a objetos, onde podemos destacar a herança de abstração, polimorfismo e principalmente delegação, sendo importante dominar estas técnicas antes de aplicar qualquer padrão de projeto. Uma excelente prática é aplicar estes padrões dentro do contexto de refatorações, ou seja, aplicar boas práticas, melhorar o design / projeto, a partir de um código existente, pois normalmente não se aplica um padrão de projeto de forma antecipada, de forma a causar complexidade desnecessária. Padrões não só são importantes para melhor a arquitetura de código existente, mas para tornar seus sistemas mais fáceis de serem modificados e evoluídos ao longo do tempo, seja para uma migração de tecnologia de acesso a dados ou user interface, ou para diferentes plataformas, como Web e Mobile.

Autor

Guinther Pauli
http://www.facebook.com/guintherpauli
http://www.gpauli.com

Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C# e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix. Most Valuable Professional (MVP) - Embarcadero Technologies.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :
www.devmedia.com.br/curso/javamagazine
(21) 3382-5038

DEVMEDIA

FireDAC: utilização avançada

Aplique mestre-detalhe, backup/restore e mais

Fique por Dentro

Neste artigo será visto a construção de uma aplicação de vendas que utiliza cadastros mestre-detalhe com a nova engine de acesso a dados do Delphi, o FireDAC. Será visto como utilizar o aplicativo do FireDAC fora do IDE do Delphi para a criação de conexões. Veremos também como utilizar o Generator do Firebird para a geração de chaves para os identificadores de tabelas, recursos estes exclusivos do Firebird/Interbase. Também serão vistos vários recursos avançados do FireDAC, como o tracing de conexões, mapeamentos de tipos de dados, tratamento de erros e recursos de Rowset Fetching para limitação de registros para tráfego de dados em redes lentas.

Ao desenvolvermos uma aplicação Client-Server com o Delphi, devemos tomar cuidado para não criarmos comandos SQL nos formulários. O Data Module foi criado justamente para concentrar os componentes de acesso a dados e todas as consultas necessárias.

O desenvolvedor pode achar mais fácil simplesmente ficar escrevendo instruções SQL nos formulários, porém, com o crescimento do sistema, sua manutenção no futuro será muito mais difícil, sendo que separando bem os formulários das queries tudo fica mais fácil.

Aplicativos que possuem estas regras de acesso a banco de dados em formulários, ferem o princípio de separação de interesses da orientação a objetos. Respeitando este princípio, temos uma maior facilidade em realizar manutenção nestes sistemas, visto que uma alteração na interface não terá nenhum efeito no Data Module e vice-versa.

A separação de interesses está intimamente ligada ao padrão de projeto MVC (**BOX 1**), muito utilizado no desenvolvimento de aplicações web. Também diz muito a respeito dos conceitos de coesão e acoplamento de classes (**BOX 2**) da orientação a objetos. Este conceito nos indica que cada parte do aplicativo deve ter uma responsabilidade, e que esta não deve se misturar com outras, como acontece com os formulários e Data Modules. Cada um tem uma responsabilidade e quando misturamos elas, estamos ferindo este importante princípio de desenvolvimento de software.

Migrando de AnyDAC para FireDAC

Para os desenvolvedores que já utilizavam o AnyDAC, existem algumas poucas diferenças para o FireDAC para considerar na hora de fazer a migração.

A primeira alteração é na API, onde foram renomeadas todas as units e também os prefixos das classes. Com o FireDAC foi implementado o mecanismo de hierarquia de namespaces (**BOX 3**).

Antes tínhamos a seguinte estrutura no AnyDAC:

- uAD<camada><função>
- Agora temos a seguinte estrutura de unit no FireDAC:
- FireDAC.<camada>.<função>.pas

Veja nos exemplos destas units:

- uADCompClient -> FireDAC.Comp.Client
- uADStanOption -> FireDAC.Stan.Option
- uADPhysIB -> FireDAC.Phys.IB

Já os nomes dos componentes foram prefixados com o FD ao invés do AD como existia anteriormente.

Para quem está em versões anteriores do Delphi com AnyDAC e deseja migrar para uma versão mais recente com FireDAC, existe um aplicativo que facilita esta migração chamado reFind.exe que se encontra na pasta Bin do diretório de instalação do Delphi.

Os pacotes em tempo de execução (Runtime packages), também foram renomeados. Ex: FireDACCommon190, FireDACCommonDriver190.

Os arquivos de inicialização também foram renomeados de ADxxx para FDxxx e tiveram uma mudança no local, agora presentes em C:\Users\Public\Documents\Embarcadero\Studio\FireDAC.

BOX 1. MVC

O padrão MVC ou Model-View-Controller, popularmente falando, pode ser considerado o ancestral de todos os demais padrões surgidos posteriormente (MVP e MVVM). Seu surgimento se deu ainda na década de 80, se tornando um padrão muito popular para a criação de aplicações Web. No MVC o Controller é definido como o ponto de entrada para as requisições de usuário, decidindo inclusive como lidar com cada entrada deste. Este mesmo Controller acessa então a camada Model, que tem a incumbência de processar a requisição, cujo resultado final é exibido na camada View.

BOX 2. Coesão e Acoplamento

O grau de Coesão na Orientação a Objetos nos indica a forma como foi projetada uma classe no que tange ao seu foco. Isso quer dizer que quando mais focada for a classe em resolver um problema, maior será a sua coesão, o que é uma coisa boa. O principal benefício da alta coesão é que essas classes são mais fáceis de manter e são menos necessárias alterações frequentes nelas do que nas de baixa coesão. Outro grande benefício é que classes com alta coesão são mais fáceis de serem reutilizadas posteriormente.

Acoplamento é o grau de conhecimento que uma classe tem sobre outra. Se o único conhecimento que a classe X tem sobre a classe Y for o que a classe Y expos através de sua interface, então a classe X e a classe Y são fracamente acopladas, o que é muito bom. Porém, se a classe A se baseia em partes da classe B que não fazem parte de sua interface, então elas estão fortemente acopladas, o que não é uma boa prática em sistemas orientados a objetos.

BOX 3. Namespaces no Delphi

O mecanismo de namespaces no Delphi foi introduzido desde a versão XE2 do IDE, principalmente devido a introdução do FireMonkey. Desta forma, podemos ter duas units com o mesmo nome final igual e namespaces diferentes, como existe na VCL e Firemonkey. Ex: Vcl.Forms, FMX.Forms. FMX.StdCtrls, Vcl.StdCtrls.

FireDAC Explorer

Para criarmos a conexão com o banco de dados, podemos fazer uso do aplicativo FireDAC Explorer, para criação do arquivo de inicialização da conexão com o banco de dados, que posteriormente será utilizado pela aplicação cliente, ou seja, a conexão e o desenvolvimento da aplicação podem ser feitos de forma separada.

O primeiro passo para efetuarmos a conexão é a abertura do arquivo INI onde estão definidas as conexões previamente (C:\Users\Public\Documents\Embarcadero\Studio\FireDAC), através do botão Open ConnDef File. Neste caminho está o arquivo de inicialização com os bancos de dados configurados para utilização no IDE do Delphi.

Em seguida, através do menu New/ConnectionDefinition iremos criar a nossa conexão para o exemplo que será desenvolvido, conforme a **Figura 1**.

As principais configurações necessárias para o exemplo na guia Definition são as seguintes:

- Database: Caminho do banco de dados;
- User_Name/Password: Usuário e senha do banco;
- MonitorBy: Forma de geração do arquivo de monitoramento de operações efetuadas;
- Protocol: Protocolo de comunicação com o banco de dados;
- Server: Número do IP do servidor de banco de dados (localhost ou 127.0.0.1 quando desejamos utilizar a própria máquina como servidor).

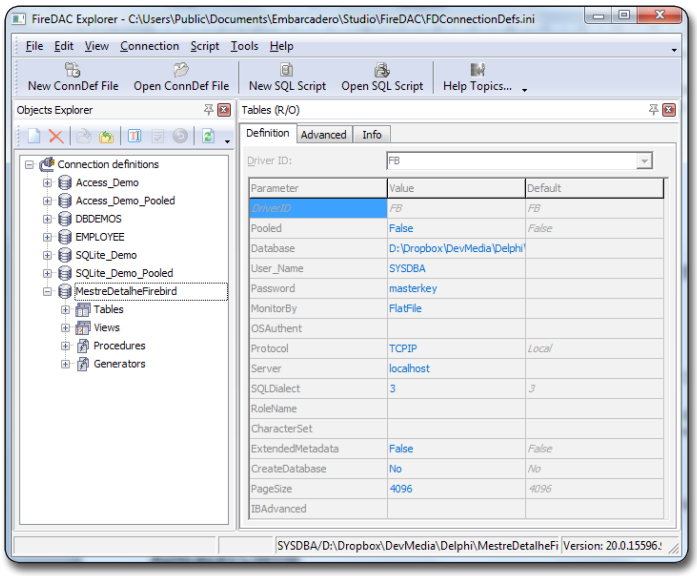


Figura 1. Conexão com o banco de dados através do FireDAC Explorer

Nota

Este aplicativo auxiliar pode ser acessado no próprio Delphi através do menu Tools/FiredacExplorer ou diretamente no executável FDAdministrator que fica na pasta Bin do diretório de instalação do Delphi ($Delphi\Bin).

Na aba Advanced existem algumas configurações avançadas da conexão, e na Info temos um resumo da configuração de forma textual.

Ao salvarmos teremos uma nova conexão pré-estabelecida e que servirá de base para configurarmos a conexão com o FireDAC na aplicação cliente posteriormente. Também temos acesso a toda base de dados configurada, suas tabelas, views, procedures e generators.

FireDAC integrado ao Delphi

Na mais recente versão do Delphi, a XE6, temos a integração do FireDAC no Data Explorer do IDE, além do dbExpress que já existia, conforme **Figura 2**. É mais uma maneira de criarmos uma conexão do FireDAC.

As novas conexões realizadas no Data Explorer do IDE serão persistidas no arquivo de inicialização localizado em /Embarcadero/Studio/FireDAC/FDConnectionDefs.ini.

Quando o sistema for instalado em outra máquina que não tenha instalado o Delphi, e em consequência disto não ter este arquivo, é necessário fazer a disponibilização deste arquivo junto do aplicativo e sua leitura em tempo de execução, para que a aplicação saiba o banco de dados que deverá utilizar, seu caminho na máquina ou rede e outras configurações do banco de dados.

Realizando tracing da conexão

O monitoramento da conexão é algo bastante interessante, principalmente quando às vezes encontramos erros difíceis de serem solucionados ou alguma parte do sistema está deixando lento

todo o sistema. Ao analisarmos um log das operações, podemos fazer uma análise profunda das operações realizadas, o tempo que levaram e muitas outras informações sobre elas.

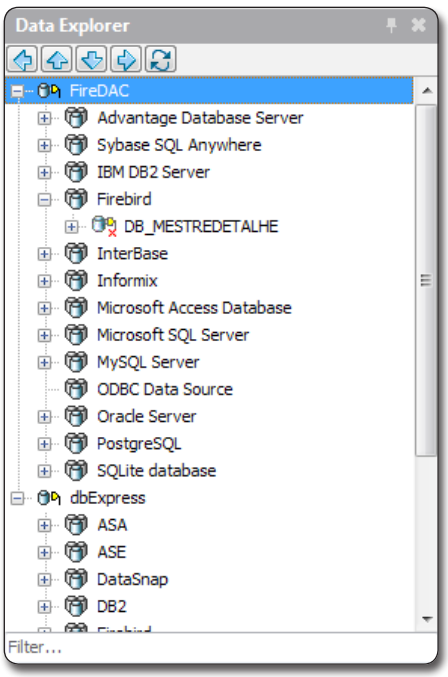


Figura 2. FireDAC integrado ao Data Explorer

Neste exemplo será feito o tracing de conexão em arquivo texto, através do componente FDMoniFlatFileClientLink, que precisa que sejam configurados alguns parâmetros para o seu funcionamento:

- FileName: local onde desejamos gravar o log;
- EventKinds: eventos passíveis de geração de logs;
- FileColumns: quais as informações que desejamos gravar no arquivo;
- FileEncoding: codificação do tipo de caracteres a ser usado no arquivo;
- ShowTrace: mostra uma mensagem de confirmação da geração do arquivo de tracing ao fecharmos o aplicativo;
- Tracing: começa a fazer a gravação ao ativarmos a opção e não faz a gravação se desabilitada.

Backup/Restore com Firebird e FireDAC

O FireDAC possui componentes que possibilitam ao desenvolvedor a realização de backup/restore de bancos de dados Firebird.

Para realização de backup utiliza-se o componente TFDFBNBackup e nele configuramos as seguintes opções:

- Link do Driver do banco de dados que está na Tool Palette FireDAC Links;
- Usuário e Senha do banco de dados;
- Host onde está hospedado o servidor de banco de dados;
- O nome do arquivo do banco de dados;
- O tipo de backup que deseja ser realizado, se é completo ou incremental;
- Protocolo de comunicação com o banco de dados, que pode ser local, TCP/IP, etc.;

Depois de configuradas estas opções, basta chamar o método Backup do componente que ele será automaticamente criado.

O recurso de Restore funciona de maneira análoga ao backup, tendo poucas diferenças de configuração. Para utilizá-lo basta adicionar o componente TFDFBNRestore e configurar as seguintes opções:

- Link do Driver do banco de dados que está na Tool Palette FireDAC Links;
- Usuário e Senha do banco de dados;
- O banco de dados que deseja-se restaurar;
- O nome dos arquivos que deseja restaurar;

Por fim, basta chamarmos o método Restore para que o banco de dados seja restaurado automaticamente.

O que temos geralmente é um menu de configuração do software restrito ao administrador do sistema, onde pode-se realizar o backup/restore pelo próprio aplicativo, sem a necessidade de copiar/excluir arquivos ou recompilar o aplicativo após a mudança, algo muito mais simples e profissional.

Nota

Este recurso de Backup/Restore não está disponível para todos os sistemas gerenciadores de bancos de dados, apenas para alguns como Firebird, Interbase, SQLite que permitem utilizar este recurso.

Mapeamento de tipos de dados com FireDAC

Aqui se encontra algumas das maiores vantagens da utilização do FireDAC no que diz respeito ao uso de vários bancos de dados por nossos aplicativos. No dia a dia nos deparamos com os mais variados tipos de clientes, grandes empresas que possuem versões pagas de bancos de dados extremamente robustos, como SQLServer, e empresas de médio e pequeno porte que preferem soluções gratuitas como Firebird.

O problema é que os bancos de dados tratam as informações de formas diferentes. De maneira mais prática, quando formos adicionar os campos TFields nos DataSets do Delphi, teremos diferentes tipos de dados e diferentes mapeamentos de um banco para o outro.

Um exemplo claro destas diferenças pode ser observado com relação ao tipo Float. No SQLServer temos um tipo de dados float que é mapeado no Delphi para o tipo TFloatField no FieldsEditor. Já no Firebird, geralmente é utilizado o tipo Numeric para pontos flutuantes, e quando adicionamos os TFields recebemos um tipo TBCDField. Desta forma, ao mudarmos o banco de dados, temos um erro em tempo de execução por essa diferença.

O FireDAC possui um mecanismo abstrato de mapeamento de tipos de dados que resolve este problema e existem duas maneiras de configurarmos seu funcionamento. Podemos criar uma regra para uma única query, ou criar um mapeamento para toda a conexão.

Para a primeira opção, basta acessarmos a opção Format-Options/MapRules. Na caixa de diálogo criamos uma nova regra de mapeamento, conforme a **Figura 3**. Na propriedade Source-DataType selecionamos o tipo de origem do banco de dados, já na propriedade TargetDataType selecionamos o tipo de dado para o qual desejamos que o tipo original seja transformado. Desta forma conseguimos padronizar os mais variados tipos de dados que podem retornar do banco de dados.

A segunda opção de mapeamento de dados é onde criamos uma regra para todas as queries que acessam a conexão. Basta configurarmos pelo Wizard na guia Options/Format Options/Data Mapping Rules, em seguida acionamos Add Rule e configuramos a mesma regra de mapeamento, conforme a **Figura 4**.

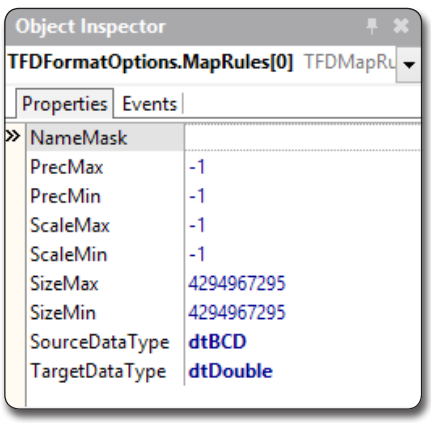


Figura 3. Mapeamento de Dados na Query

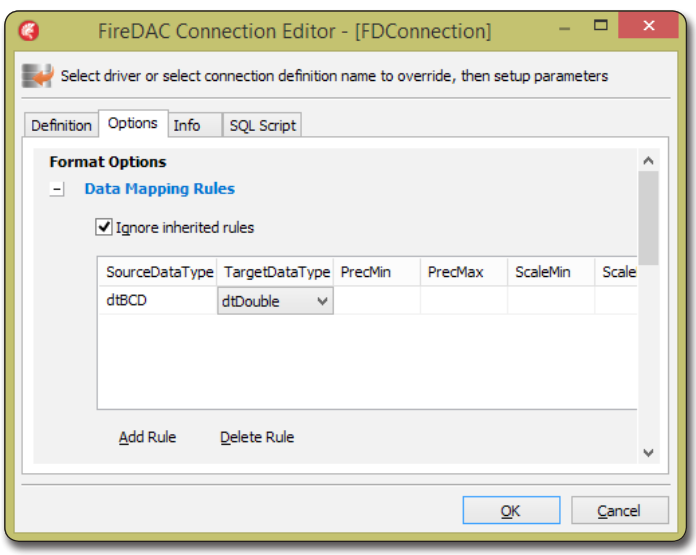


Figura 4. Mapeamento de Dados na Conexão

Desenvolvimento do projeto exemplo

Para o desenvolvimento da aplicação de exemplo, criamos um novo projeto VCL e adicionamos também um Data Module ao mesmo.

O cenário utilizado neste exemplo prático talvez seja o mais comum dos desenvolvedores Delphi, que é uma aplicação de vendas, onde temos uma relação mestre-detalhe entre a Venda e os Itens da Venda.

Banco de Dados

O banco de dados usado no exemplo será o Firebird, por ser o mais utilizado pela comunidade de desenvolvedores Delphi. Veja na **Listagem 1** o script de criação da sua estrutura.

Listagem 1. Script de criação das tabelas do banco de dados

```
01 CREATE TABLE CLIENTES (
02  ID_CLIENTE INTEGER NOT NULL,
03  NOME_CLIENTE VARCHAR(50) NOT NULL
04 );
05 ALTER TABLE CLIENTES ADD CONSTRAINT PK_CLIENTES
    PRIMARY KEY (ID_CLIENTE);
06 CREATE TABLE PRODUTOS (
07  ID_PRODUTO INTEGER NOT NULL,
08  DESCRICAO VARCHAR(50) NOT NULL,
09  VALOR_UNITARIO NUMERIC(10,2) NOT NULL
10 );
11 ALTER TABLE PRODUTOS ADD CONSTRAINT PK_PRODUTOS
    PRIMARY KEY (ID_PRODUTO);
12 CREATE TABLE VENDAS (
13  ID_VENDA INTEGER NOT NULL,
14  ID_CLIENTE INTEGER NOT NULL,
15  DATA_VENDA DATE NOT NULL,
16  VALOR_TOTAL NUMERIC(10,2)
17 );
18 ALTER TABLE VENDAS ADD CONSTRAINT PK_VENDAS
    PRIMARY KEY (ID_VENDA);
19 ALTER TABLE VENDAS ADD CONSTRAINT FK_VENDAS_1
     FOREIGN KEY (ID_CLIENTE)
     REFERENCES CLIENTES (ID_CLIENTE);
20 CREATE TABLE ITENS_VENDA (
21  ID_VENDA INTEGER NOT NULL,
22  ID_PRODUTO INTEGER NOT NULL,
23  QUANTIDADE INTEGER NOT NULL,
24  VALOR_UNITARIO NUMERIC(10,2) NOT NULL
25 );
26 ALTER TABLE ITENS_VENDA ADD CONSTRAINT PK_ITENS_VENDA
    PRIMARY KEY (ID_VENDA, ID_PRODUTO);
27 ALTER TABLE ITENS_VENDA ADD CONSTRAINT FK_ITENS_VENDA_1
    FOREIGN KEY (ID_VENDA) REFERENCES VENDAS (ID_VENDA);
28 ALTER TABLE ITENS_VENDA ADD CONSTRAINT FK_ITENS_VENDA_2
    FOREIGN KEY (ID_PRODUTO) REFERENCES PRODUTOS (ID_PRODUTO);
29 CREATE SEQUENCE GEN_CLIENTES_ID;
30 CREATE SEQUENCE  GEN_PRODUTOS_ID;
31 CREATE SEQUENCE  GEN_VENDAS_ID;
```

O detalhe da criação das tabelas não será explicado por não estar no escopo deste artigo. Destaque apenas pela tabela de Itens da Venda, nela temos dois campos chaves primárias (IdVenda, IdProduto), neste caso precisamos destas duas informações para identificar unicamente o registro na tabela itens. Ambos os campos também são chave estrangeira, para termos a verificação de integridade dos dados ao adicionarmos/alterarmos um registro. Esta forma de criação da tabela com dois campos formando a chave primária é criticada por alguns profissionais de informática, como os administradores de banco de dados que dizem que esta abordagem fere as regras da normalização de dados porém reduz o número de colunas da tabela. Se fossemos normalizar a tabela teríamos que adicionar mais um campo na tabela de Itens,

simplesmente para criar o auto incremento, sendo que já existem campos na tabela que diferenciam os registros unicamente.

Foram criadas três sequências para as tabelas Clientes, Produtos e Vendas que serão utilizadas pelo FireDAC na aplicação cliente, ou seja, não teremos triggers/store procedures para efetuar o auto incremento.

Aplicação cliente

Para estabelecermos a conexão com o banco de dados criado, faremos uso do componente FDConnection. Não será necessária a configuração de nenhum parâmetro, basta escolhermos a configuração pré-definida no FireDAC Explorer e o componente já estará apto a ser utilizado, conforme a **Figura 5**.

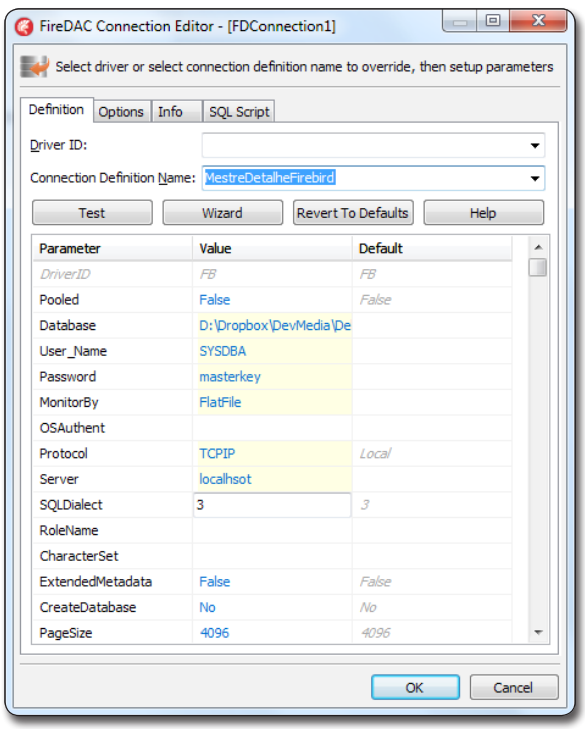


Figura 5. Utilizando a conexão estabelecida no FireDAC Explore

Existem dois detalhes a serem observados antes de começarmos a escrever as queries. Primeiramente devemos adicionar o link do driver para o banco de dados Firebird, através do componente FDPhysFBDriverLink, que adiciona uma unit na cláusula uses do Data Module. O segundo detalhe a ser cuidado é a necessidade de prover um cursor para o banco de dados ao ser utilizada a aplicação para a navegação nos registros. Para isso basta adicionar no próprio Data Module o componente FDGUIxWaitCursor.

Na **Figura 6** temos a interface da aplicação cliente. Para desenvolvermos a barra de ferramentas foi utilizado o componente da VCL chamado TCategoryButtons. Dentro dele criamos categorias (TButtonCategory) e dentro destas categorias criamos os itens (TButtonItem). Cada um destes itens pode ser vinculado a uma ação no componente TActionList, herdando todas as propriedades das ações, como o Caption, Image, atalho e etc.

Nota

O componente FDGUIxWaitCursor possui uma propriedade interessante chamada ScreenCursor, que nos permite escolher como queremos que fique o cursor do mouse no momento que o FireDAC processa uma instrução SQL.

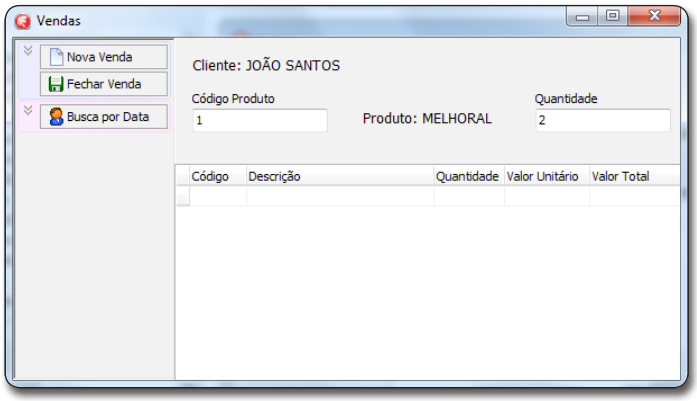


Figura 6. Interface da aplicação cliente

Configurando o relacionamento Master-Detail

Para a realização do relacionamento Master-Detail, configuramos dois componentes FDQuery (FDQryVenda, FDQryItens), conforme a **Listagem 2**.

Listagem 2. Query de Venda e Itens

```
01 select *
02 from VENDAS
03 where ID_VENDA = :ID_VENDA

04 select *
05 from ITENS_VENDA
06 where ID_VENDA = :ID_VENDA
```

Na primeira query nós buscamos os dados da venda e na segunda temos a chave estrangeira da tabela de itens apontando para um parâmetro com o mesmo nome da chave primária da tabela de venda. Estas são consultas bem comuns quando estamos trabalhando com relacionamentos Master-Detail. Após isso configuramos parâmetro IdVenda, como parâmetro de entrada e tipo Integer na guia Parameters. Mas apenas isso ainda não basta para realizarmos o relacionamento, devemos adicionar tambémum componente DataSource apontando para a query de vendas. Em seguida, na query de itens configuramos a propriedade MasterSource apontando para o DataSource da query master FDQryVenda e a propriedade MasterFields para IdVenda. Em seguida, na propriedade DetailFields da query detalhe, informamos o campo IdVenda novamente. Por fim, configuramos a propriedade IndexFieldNames, que é o campo que será indexado da tabela de detalhes, que também será o campo IdVenda. Com esta configuração já podemos visualizar os dados executando o aplicativo.

Se executarmos a aplicação neste momento, teremos o mestre-detalhe funcionando em tempo real, ou seja, no momento que adicionamos a venda, ela já é persistida no banco de dados.

Da mesma forma ocorre com os itens, a cada um deles que é adicionado, também é persistido em tempo real.

Para muitos a forma ideal é a forma como estávamos acostumados com o dbExpress, onde tudo era realizado de forma local em memória, para depois ser persistido numa única requisição, aplicando todos de uma única vez. Isso também é possível de ser feito com o FireDAC, não é necessário termos um ClientDataSet, pois o FDQuery também possui um mecanismo de cache. Para habilitarmos, basta ativar a propriedade CachedUpdates do FDQryVenda e FDQryItens.

Na query de detalhe FDQryItens, precisamos realizar mais algumas configurações. A primeira delas é ativar as propriedades FetchOptions/Cache/fiDetails e FetchOptions/DetailCascade.

Neste momento, as informações não serão mais persistidas no banco de dados automaticamente em tempo real como anteriormente, mas sim somente em cache, aguardando um ApplyUpdates para serem persistidas.

Porém em nossos Fields não temos DataSets aninhados e em nossa query FDQryVenda não temos aquele campo do tipo TDataSet, como tínhamos no ClientDataSet, desta forma, ao chamarmos um ApplyUpdates, a alteração não será refletida na query detalhe ainda.

Para resolver este problema, primeiramente precisamos de um novo componente chamado TFDSchemaAdapter para realizar o cache de ambos os DataSets. Agora basta apontar a propriedade SchemaAdapter tanto da query mestre quanto a detalhe para o componente FDSchemaAdapter adicionado.

Nota

O componente TFDSchemaAdapter tem a função de concentrar os caches dos DataSets. Usado quando necessitamos efetuar um relacionamento mestre-detalhe. Um único componente permite concentrar o cache de qualquer número de DataSets.

Agora basta em nossa ação "Fechar Venda" chamarmos o método ApplyUpdates do FDSchemaAdapter e não da query master como estávamos acostumados anteriormente, conforme a **Listagem 3**. Pronto, já temos tanto a venda quanto os itens persistidos no banco, porém ainda temos os dados tanto do FDQryVenda quanto FDQryItens em memória, isto porque o FDSchemaAdapter não limpa o Delta dos DataSets, de forma que se chamarmos novamente o ApplyUptades os dados serão gravados novamente, mesmo dando um Refresh nas queries.

Para resolver este problema, devemos executar o método CommitUpdates em cada uma das queries após ser efetuado o ApplyUptates. Poderíamos chamar esses métodos diretamente na ação "Fechar Venda", após o ApplyUpdates, porém, o componente FDSchemaAdapter possui um evento chamado AfterApplyUpdate, e é nele que iremos realizar o Commit nas duas queries, conforme a **Listagem 3**.

Com isso temos nosso relacionamento mestre-detalhe em pleno funcionamento, com os caches das consultas sendo atualizados a cada envio para o banco de dados.

Listagem 3. Ação Fechar Venda do formulário de vendas e Evento AfterApplyUpdate do FDSchemaAdapter

```
01 procedure TfrmVenda.actFecharVendaExecute(Sender: TObject);
02 begin
03   DM.FDSchemaAdapter.ApplyUpdates();
04 end;
05
06 procedure TDM.FDSchemaAdapterAfterApplyUpdate(Sender: TObject);
07 begin
08   FDQryVenda.CommitUpdates();
09   FDQryItens.CommitUpdates();
10 end;
```

Utilizando evento OnUpdateRecord do FDQuery para geração de chaves

O Firebird, até a versão 2.5.x, não possui um tipo de dado nativo de auto incremento como o SQL Server por exemplo, porém, existem outras maneiras de trabalharmos com esse tipo de campo. Pode ser através de triggers, stored procedures ou programaticamente antes de fazermos a execução do comando insert na aplicação.

Neste exemplo será utilizada a terceira abordagem. Para isso, foi criado um método na seção private do DataModule chamado GeneratorId. Ele recebe um parâmetro que é o nome da sequence no banco de dados, neste caso GEN_VENDAS_ID.

Na implementação do método temos a criação da query em tempo de execução, onde é passada a conexão e qual o SQL responsável por buscar o próximo ID do Firebird, juntamente com o nome da sequence. Na sequência, a query é aberta e o ID retornado, para liberá-la da memória a seguir.

O evento OnUpdateRecord do componente FDQuery do FireDAC ocorre momentos antes do DataSet enviar a instrução com as modificações para o banco de dados. O usuário não pode mais modificar nenhum dado neste momento.

Usamos este evento para fazer a modificação de algum campo ou atribuir algum valor a algum campo antes dele ser aplicado ao banco, conforme mostra a **Listagem 4**. Outro uso deste recurso é quando precisamos fazer alguma operação em outra tabela antes de aplicar estas alterações no banco de dados.

Este evento é muito parecido com o evento OnUpdateRecord que existe no componente DataSetProvider, quando utilizamos dbExpress ou outra Engine de acesso a dados.

Ao criarmos este evento, os seguintes parâmetros são passados pra ele:

- ASender: DataSet que está fazendo a modificação;
- ARequest: tipo de requisição que está sendo feita (inserção, atualização, deleção);
- AAction: qual ação deve acontecer no caso de haver um erro;
- AOptions: algumas opções avançadas que existem no FireDAC.

Na **Listagem 4** temos a implementação do evento OnUpdateRecord do componente FDQuery. Nele primeiramente verificamos se o tipo de requisição é de insert. Se for uma inserção, é realizada a edição do DataSet que é retornado pelo parâmetro, passando

o gerador provido pelo método GeneratorId e confirmando a operação através do Post.

O problema é que o componente FDQuery não consegue processar esta informação diretamente, por isso, precisamos adicionar um componente FDUpdateSQL, que irá fazer este processamento com os dados atualizados.

Listagem 4. Evento OnUpdateRecord e método de geração de identificadores

```
01 procedure TDM.FDQryVendaUpdateRecord(ASender: TDataSet;
02  ARequest: TFDUpdateRequest; var AAction: TFDErrorAction;
03  AOptions: TFDUpdateRowOptions);
04 begin
05  if ARequest = arInsert then
06  begin
07   ASender.Edit;
08   ASender.FieldByName('ID_VENDA').NewValue :=
     GeneratorId('GEN_VENDAS_ID');
09   ASender.Post;
10  end;
11  FDUpdateSQLVenda.DataSet := (ASender as TFDAdaptedDataSet);
12  FDUpdateSQLVenda.Apply(ARequest, AAction, AOptions);
13  AAction := eaApplied;
14 end;
15
16 function TDM.GeneratorId(const AName: string): Integer;
17 var
18  FDQuery: TFDQuery;
19 begin
20  FDQuery := TFDQuery.Create(nil);
21  try
22   FDQuery.Connection := FDConnection;
23   FDQuery.SQL.Add('SELECT NEXT VALUE FOR ' + AName +
     ' FROM RDB$DATABASE');
24   FDQuery.Open();
25   Result := FDQuery.Fields[0].AsInteger;
26  finally
27   FDQuery.Free;
28  end;
29 end;
```

A configuração deste componente é muito simples, basta clicarmos duas vezes sobre ele, escolhermos a tabela que desejamos fazer as operações e executar a ação "Generate SQL". Após isso, temos as instruções SQL prontas para serem utilizadas na guia SQL Commands, conforme a **Figura 6**.

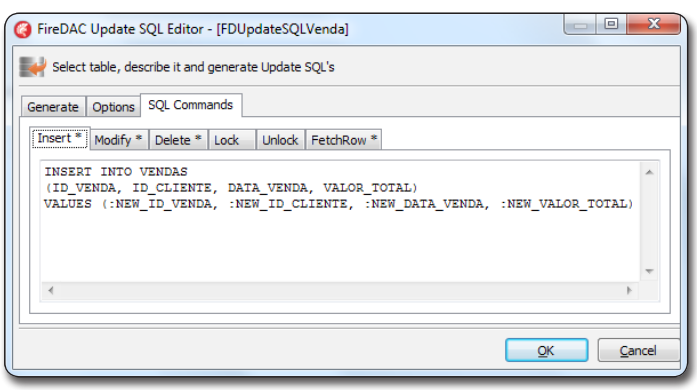


Figura 6. Comandos gerados pelo componente FDUpdateSQL

O que fazemos a partir da linha 11 da **Listagem 4** é a substituição do SQL gerado pelo FDQuery pelo que configuramos no componente SQLUpdate, recebendo o DataPacket contido no FDQuery e aplicando este comando no banco de dados através do Apply, repassando os parâmetros do evento para o Apply. Por último, só informamos ao parâmetro AAction que o processamento foi efetuado.

Configurando Auto-Incremento automático do FireDAC

O FireDAC nos possibilita configurar o auto incremento automático através dos generators do Firebird, sem necessidade de utilizar triggers, stored procedures ou qualquer outro mecanismo de busca do próximo Id, como na abordagem do tópico anterior.

Para configurá-lo, basta entrarmos no Wizard da query (em Options) escolhendo Update Options/Automatic Incrementing. Basta então configurarmos duas opções Generator e Autoinc Fields. Em Generator, basta informarmos o nome da sequence criada no Firebird e em Autoinc Fields escolhemos o campo chave que desejamos que seja configurado o auto incremento, conforme **Figura 7**.

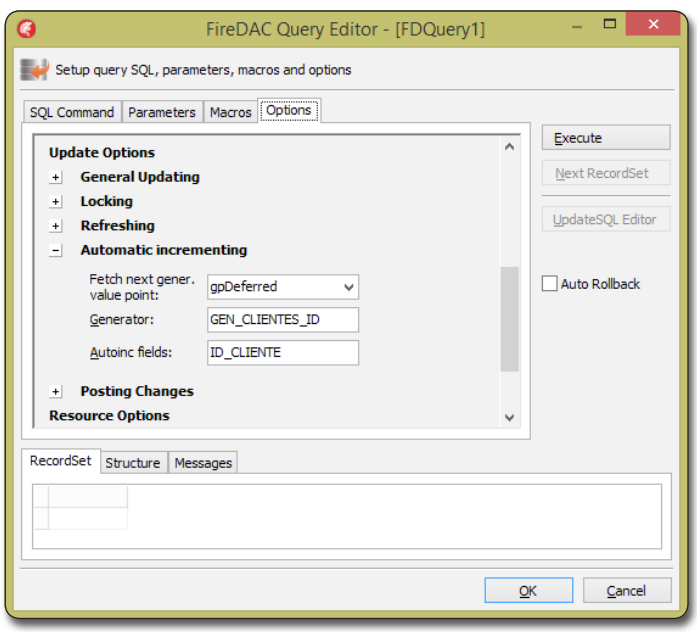


Figura 7. Auto incremento do Firebird pelo FireDAC

Tratamento de erros com FireDAC

Todos sabemos que não existe como nos livrarmos das exceções, seja por algum descuido de desenvolvimento ou por algo indevido informado pelo usuário, sempre teremos situações onde teremos a necessidade de trabalharmos com isso.

Para isso, iremos trabalhar com o evento chamado OnUpdateError (**Listagem 5**) dos componentes FDQuery. Neste evento também teremos alguns parâmetros, são eles:

- ASender: DataSet que ocorreu o erro;
- AException: é a exceção que ocorreu;
- ARow: representa a linha que ocorreu a exceção;

• ARequest: tipo de requisição que gerou a exceção (insert, update, delete, etc);
• AAction: qual ação deve acontecer no caso de haver um erro;

Listagem 5. Evento OnUpdateError

```
01 procedure TDM.FDQryItensUpdateError(ASender: TDataSet; AException:
   EFDException;
02  ARow: TFDDatSRow; ARequest: TFDUpdateRequest;
   var AAction: TFDErrorAction);
03 begin
04  if AException is EFDDBEngineException then
05  begin
06   if EFDDBEngineException(AException).Kind =
     TFDCommandExceptionKind.ekFKViolated then
07   begin
08    raise Exception.Create('Produtos iguais para a mesma venda!' + #13 +
09     'Código do Produto: ' + ARow.GetData('ID_PRODUTO'));
10   end;
11  end;
12 end;
```

Neste exemplo de utilização (**Listagem 5**), verificamos se o tipo de erro é de banco de dados, conforme a linha 4. Caso seja, realizamos um typecast da exception para EFDDBEngineException, que através da propriedade Kind nos proporciona uma grande quantidade de erros já mapeados pelo FireDAC.

Neste caso, verificamos se o tipo de erro corresponde ao ekFKViolated na query de itens, o que significa que se o usuário informar duas vezes o mesmo produto para a mesma venda, teremos um erro de PK, o que fará com que o compilador entre no if (linha 06). Se isso ocorrer, geramos uma exceção com a mensagem de erro e ainda através do parâmetro ARow, conseguimos exibir qual o código do produto duplicado para o usuário para que ele possa fazer o ajuste necessário e, em seguida, tentar novamente realizar a persistência dos dados.

O componente FDQuery do FireDAC também possui o evento OnReconcileError onde podemos exibir uma mensagem de erro para o usuário através do menu File/New/Other/VCL Reconcile Error Dialog (assim como no ClientDataSet).

Busca do Cliente

Para criarmos a busca pelo cliente no momento da venda, por se tratar de uma tabela de poucos registros, faremos uso do componente FDTable, que é um componente para navegação e edição diretamente na tabela, permitindo o uso bidirecional.

Sua configuração é simples, basta informarmos o nome da tabela que desejamos carregar na propriedade TableName.

Na tela de busca de vendas, teremos um grid onde serão carregados todos os clientes para seleção pelo usuário e ao acionarmos o duplo clique, este IdCliente selecionado será devolvido ao formulário principal para ser atribuído ao FDQryVenda, conforme a **Listagem 6**.

Note no código da **Listagem 6** que as primeiras 14 linhas referem-se à implementação do método BuscaCliente. Na linha 03 o formulário é criado e posteriormente chamado (linha 05). A seguir, na linha 06, é verificado se existe registro selecionado, como o próprio método do formulário sugere. Caso haja, o valor selecionado é atribuído ao ID_Cilente, realizando o Post em seguida (linhas 08 e 09). Na linha 12 o formulário é liberado da memória. Na **Listagem 7** temos o evento OnClick do Grid (formulário de busca do Cliente), onde simplesmente alteramos o flag do campo FRegistroSelcionado para True e atribuímos o valor do campo ao FIdSelecionado (linhas 03 e 04). A seguir o método close é chamado, encerrando a tela (linha 05).

Listagem 6. Busca e Posicionamento do Cliente

```
01 procedure TfrmVenda.BuscaCliente;
02 begin
03  Application.CreateForm(TfrmBuscaCliente, frmBuscaCliente);
04  try
05   frmBuscaCliente.ShowModal;
06   if frmBuscaCliente.RegistroSelecionado then
07   begin
08    DM.FDQryVenda.FieldByName('ID_CLIENTE').AsInteger :=
     frmBuscaCliente.IdSelecionado;
09    DM.FDQryVenda.Post;
10   end;
11  finally
12   frmBuscaCliente.Release;
13  end;
14 end;
```

Listagem 7. Evento de duplo clique sobre o Grid

```
01 procedure TfrmBuscaCliente.grdClientesDblClick(Sender: TObject);
02 begin
03  FRegistroSelecionado := True;
04  FIdSelecionado := dsCliente.DataSet.Fields[0].AsInteger;
05  Close;
06 end;
```

Busca do Produto

No exemplo iremos realizar a busca do produto com outro recurso do FireDAC que é a execução de um SQL diretamente no componente FDConnection. No formulário principal, quando o usuário digita o código do produto e pressiona enter, não teremos um FDQuery para realizar esta busca.

Observe que este é o momento onde o desenvolvedor sente-se tentado a executar o SQL no evento OnKeyPress diretamente, porém, nós não iremos fazer desta maneira, pois o lugar da instrução SQL é no DataModule, e não no formulário, como já sabemos. Escrever SQL nos formulários fere as regras de diversos padrões de projeto.

Na **Listagem 8** é ilustrado o método público BuscaProdutoByCod no DataModule, passando o valor informado pelo usuário no Edit e retornando o nome do produto caso encontrado. Caso contrário, será gerada uma exceção para o usuário. O método que possibilita fazer esta busca é o ExecSQLScalar, que nos retorna um tipo Variant, que pode ser qualquer tipo de informação. No nosso caso, será o IdProduto para atribuirmos à coluna IdProduto da tabela Itens da Venda.

Na **Listagem 9** temos a descrição e implementação da classe EProdutoNaoEncontrado que é responsável por gerar uma exceção personalizada para o usuário, conforme mencionado anteriormente.

Listagem 8. Método BuscaProdutoByCod do DataModule

```
01 function TDM.BuscaProdutoByCod(const ACodigo: string): integer;
02 const
03   SQL = 'select ID_PRODUTO from PRODUTOS where CODIGO = %s';
04 var
05   IdProduto: Variant;
06 begin
07   IdProduto := FDConnection.ExecSQLScalar(Format(SQL, [ACodigo]));
08   if IdProduto = Null then
09     raise EProdutoNaoEncontrado.Create(ACodigo)
10   else Result := IdProduto;
11 end;
```

Listagem 9. Classe de Exceção Personalizada

```
01 // classe de exceção personalizada
02 EProdutoNaoEncontrado = class(EArgumentException)
03   constructor Create(ACodigo: string);
04 end;

05 constructor EProdutoNaoEncontrado.Create(ACodigo: string);
06 begin
07   inherited CreateFmt('O código %s não foi encontrado', [ACodigo]);
08 end;
```

Nota

EArgumentException é o tipo de exceção que deve ser gerada quando recebemos algum argumento/parâmetro inválido. Deve ser levantada quando não esperamos este parâmetro, facilitando o Debug. Desenvolvedores geralmente criam exceções do tipo Exception, a mais genérica de todas, quando temos diversas classes de tratamento de exceções disponíveis na API do Delphi, ainda assim sendo possível criar nossas próprias exceções personalizadas.

Observe que a forma como desenvolvemos nosso método facilita muito o trabalho dos formulários, pois não fazemos nenhum tipo de validação de dados nele, simplesmente coletamos as informações e passamos como parâmetro para as queries e métodos do DataModule. Note como ficou a implementação de busca do produto no evento OnKeyPress do Edit (**Listagem 10**).

Primeiramente é verificado se foi pressionada a tecla enter (linha 05). A seguir, o código do produto é atribuído de acordo com sua descrição, passada por parâmetro ao método BuscaProdutoByCod do DataModule (linha 08). O ID_Produto recebe o valor atribuído à variável IdProduto. Caso não seja encontrado, assim como mencionado anteriormente, a nossa classe EProdutoNaoEncontrado é acionada, exibindo a mensagem de erro (linhas 10 a 13).

Busca de Vendas

Para completar a manutenção de vendas, precisamos de uma tela de consulta por período, afim de recuperamos as informações cadastradas. Para isso, adicionamos um componente FDQuery e a instrução referente a consulta, conforme a **Listagem 11**.

Listagem 10. Buscando o Produto – evento OnKeyPress do edtCodigo

```
01 procedure TfrmVenda.edtCodigoKeyPress(Sender: TObject; var Key: Char);
02 var
03   IdProduto: Integer;
04 begin
05   if Key = #13 then
06   begin
07     try
08       IdProduto := DM.BuscaProdutoByCod(edtCodigo.Text);
09       DM.FDQryItens.FieldByName('ID_PRODUTO').AsInteger := IdProduto;
10     except
11       on E: EProdutoNaoEncontrado do
12         MessageDlg(E.Message, mtWarning, [mbOK], 0);
13     end;
14   end;
15 end;
```

Listagem 11. SQL consulta de vendas

```
01 select V.*, C.NOME_CLIENTE
02 from VENDAS V
03 inner join CLIENTES C on (V.ID_CLIENTE = C.ID_CLIENTE)
04 where DATA_VENDA between :DT_INICIO and :DT_FIM
```

A consulta busca todos os dados da tabela VENDAS e realiza um Join com a tabela CLIENTES para exibir também o nome do cliente, facilitando a localização do registro.

Na **Listagem 12** apresentamos o código da ação BuscaPorData, localizada no formulário de vendas. Nesta ação temos a criação do formulário de pesquisa em tempo de execução através da variável global Application (linha 03), que pode ser utilizada em todos os formulários do projeto. Na sequência, chamamos o formulário de pesquisa através do método ShowModal (linha 05). Após efetuada a pesquisa, no retorno testamos a propriedade RegistroSelecionado (linha 06), que assim como na ação anterior (BuscaCliente – **Listagem 6**), nos indica se foi escolhido algum registro na tela de pesquisa. Confirmada a seleção, o componente FDQryVenda recebe como parâmetro o IdSelecionado (linha 09), abrindo-o posteriormente (linha 10). No final, liberamos a instância do formulário de pesquisa da memória através do método Release (linha 13).

A medida que as informações dos sistemas crescem, devemos tomar cuidado com as consultas de busca, pois quando não são bem parametrizadas, podem retornar muitos registros, deixando o software lento, principalmente quando há problemas de infraestrutura relacionados à rede.

Para resolver este tipo de problema, existe um recurso no FireDAC que permite otimizar as consultas de busca de registros, limitando o número de linhas que serão carregadas pela Query independente dos parâmetros passados. Este é o recurso de Rowsert Fetch, que permite realizar o controle de requisições ao banco de dados.

Para isso, basta acessar a guia Options do Wizard de criação da query de consulta, conforme ilustra a **Figura 8**. Ao escolhermos a opção Fetch Options, basta informar o número máximo de registros a retornar, no item Maximum Records to Fetch. Por padrão, o valor informado é de -1, que significa que deve trazer

todos. Neste caso, modificamos pra 100 no exemplo, desta forma, não precisamos nos preocupar se usuário informar um intervalo muito grande entre as datas de pesquisa, tendo a certeza que não teremos nenhum travamento do sistema por esse motivo.

Listagem 12. Ação que cria o formulário e posiciona registro

```
01 procedure TfrmVenda.actBuscaPorDataExecute(Sender: TObject);
02 begin
03   Application.CreateForm(TfrmBuscaVendas, frmBuscaVendas);
04   try
05     frmBuscaVendas.ShowModal;
06     if frmBuscaVendas.RegistroSelecionado then
07     begin
08       DM.FDQryVenda.Close;
09       DM.FDQryVenda.Params[0].AsInteger := frmBuscaVendas.IdSelecionado;
10       DM.FDQryVenda.Open();
11     end;
12   finally
13     frmBuscaVendas.Release;
14   end;
15 end;
```

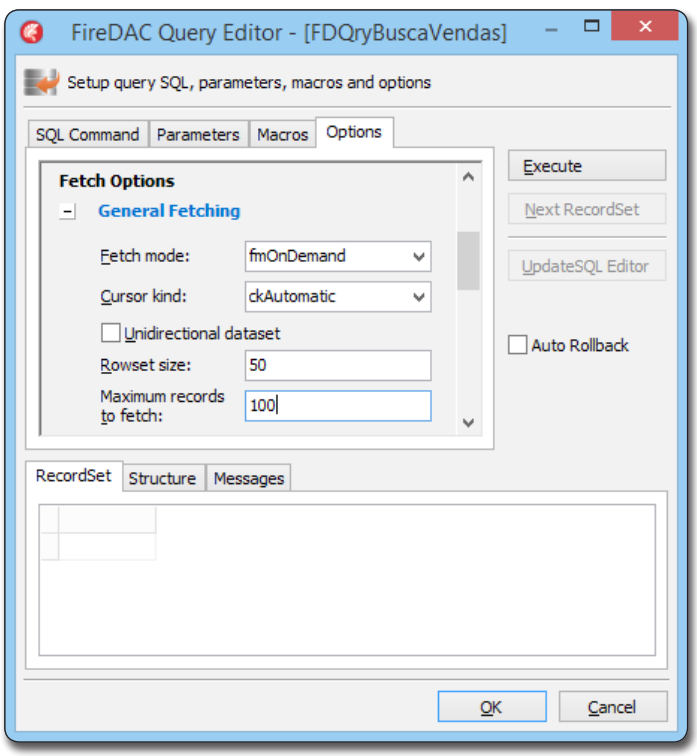


Figura 8. Configurando o número máximo de registros

O FireDAC nos possibilita trabalhar de forma semelhante ao que já estávamos acostumados com outras Engines de acesso a dados, como o BDE e o dbExpress. Isso porque também possui os DataSets e Fields em sua essência. O Delphi XE6 junto com FireDAC é sem dúvida o que existe de melhor para desenvolver aplicações Client-Server no Delphi. Contudo, ainda há a possibilidade de se integrar ao DataSnap.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi (Desktop) e C# (Web), utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Links:

Download Delphi XE6 Trial
https://www.embarcadero.com/br/products/delphi

Embarcadero Developer Network
http://edn.embarcadero.com/

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum

DEVMEDIA

Princípios SOLID com Delphi – Parte 1

Aprenda a criar suas classes obedecendo estes importantes princípios da orientação a objetos

Este artigo faz parte de um curso

Fique por Dentro

Neste artigo veremos os princípios SOLID da orientação a objetos que permitem que nossos softwares sejam flexíveis, de fácil manutenção e entendimento do código. Na primeira parte do artigo aprenderemos a aplicar os três dos cinco princípios SOLID (Responsabilidade Única, Aberto/Fechado, Substituição de Liskov com o Delphi em situações bem comuns do dia a dia de todo desenvolvedor. Veremos as vantagens e as maneiras corretas de aplicar este princípio passo a passo. Ainda veremos como unir o desenvolvimento RAD do Delphi com boas práticas de orientação a objetos.

Muitos desenvolvedores Delphi não seguem bons princípios de orientação a objetos com a justificativa ou desculpa como eu prefiro dizer que a Linguagem/IDE/Compilador não oferece bons frameworks para trabalhar desta maneira.

Na verdade o que falta para esses desenvolvedores é abrir a mente para um pensamento orientado a objetos como fazem desenvolvedores Java e C#, pois o Delphi é uma linguagem multiparadigma, suporta tanto a programação procedural quanto a orientada a objetos, cabe a nós desenvolvedores escolhermos de que maneira preferimos trabalhar. É claro que a adição de frameworks de mapeamento objeto-relacional ou injeção de dependências facilitaria a vida dos desenvolvedores e a Embarcadero poderia focar um pouco mais neste quesito.

É perfeitamente possível unir toda a produtividade que o Delphi nos oferece por ser uma ferramenta RAD com bons princípios de orientação a objetos e isto ficará muito claro no decorrer do artigo.

Alguns desenvolvedores acreditam que mesmo quando estamos somente adicionando e manipulando componentes em um formulário, estamos trabalhando orientado a objetos, pois cada novo formulário que criamos no Delphi gera uma nova classe que herda de TForm e ao adicionar componentes no formulário estamos adicionando objetos no formulário, pois cada componente representa uma classe que herda em última instância da classe TComponent.

SOLID

No desenvolvimento orientado a objetos existem alguns princípios que todo o bom desenvolvedor de sistemas deve adotar para ter um sistema com um código mais limpo e de fácil entendimento e manutenção, e quando exista a necessidade de manutenção ela seja rápida e concisa, de forma que não cause nenhum impacto em outros módulos, sendo mais evolutivas e flexíveis.

SOLID são cinco princípios básicos da orientação a objetos que trazem inúmeros benefícios em uma modelagem de classes. Estes princípios foram base para muitos dos padrões de projeto existentes nos dias de hoje. Cada letra representa um princípio, são eles:

- S – Single Responsibility Principle (SRP): princípio da responsabilidade única;
- O – Open Closed Principle (OCP): princípio do aberto/fechado;
- L – Liskov Substituition Principle (LSP): princípio da substituição de Liskov;
- I – Inteface Segregation Principle (ISP): princípio da segregação de interfaces;
- D – Dependency Inversion Principle (DIP): princípio da inversão de dependência;

Serão tratados cada um dos princípios de forma individual, os três primeiros nesta primeira parte do artigo e os dois últimos na segunda parte do artigo, na próxima edição da revista ClubeDelphi.

Quando seguimos estes princípios da orientação a objetos conseguimos ter uma modelagem de classes bastante correta, de forma que os sistemas não tenham que sofrer muitas alterações nas classes já existentes no futuro.

É indiscutível que os princípios SOLID são base para o desenvolvimento de qualquer aplicação que deseje seguir o paradigma orientado a objetos.

SRP (Sigle Responsibility Principle)

Este princípio de responsabilidade única da orientação a objetos nos ensina que nossas classes devem ter somente uma responsabilidade dentro do modelo e deve estar completamente encapsulada em seus métodos. Todos os métodos contidos nas classes, tanto os privados, protegidos ou públicos devem agir de modo a se completarem e atingir o objetivo que é único desta classe.

Quando seguimos este princípio da maneira descrita acima, esta classe terá somente uma razão para ser alterada. Esta classe poderá ainda conter vários métodos sendo que os mesmos contribuam para somente aquela única tarefa proposta para a classe, também não teremos muitas dependências de outros módulos, de maneira que ao alterarmos esta classe não será necessário recompilar outras classes do sistema.

Quando construímos classes que seguem o princípio de responsabilidade única podemos ter a certeza que estamos criando um sistema que tem um baixo acoplamento **(BOX 1)** entre os módulos.

BOX 1. Acoplamento

Alto acoplamento é nome dado à ocorrência de alto grau de dependência entre partes de um software, sendo que estas partes deveriam funcionar de forma independente. Numa situação real, do dia a dia poderíamos pensar da seguinte maneira: para se conectar a um banco de dados para realizar uma atividade possuímos um método que realiza a conexão com o banco. No interior de seu método de conexão, existe a chamada para outro método, sendo que este último apenas identifica se ocorreu algum erro e registra em um arquivo ou tabela de logs do sistema. Assim, sabemos que existe uma dependência explícita do método de conexão para com o método de registro de log. Um só pode funcionar se o outro funcionar. Tal dependência é chamada de acoplamento. Se este fato ocorre de forma sistemática no projeto de software, dizemos que a aplicação é altamente acoplada.

Este princípio nos ensina que nossas classes devem ter uma única tarefa em nossa aplicação, do contrário significa que ao certo devíamos dividir esta classe em duas ou mais classes para seguir o princípio, porque cada tarefa é uma fonte de mudanças no sistema e estas tarefas por isso devem estar bem isoladas de forma que uma tarefa não influa em outras. É um princípio bastante simples e que devemos adotar para termos um sistema que demande facilidade de manutenção no futuro.

OCP – Open Closed Principle

Este princípio do Aberto/Fechado da orientação a objetos é um dos que mais chama a atenção dos estudiosos de arquitetura de sistemas, ele nos traz uma definição muito interessante que devemos tomar cuidado em nossas classes, no diz que elas devem estar abertas para extensão e fechadas para modificação, ou seja, quando criamos uma classe devemos levar em consideração uma codificação que muito dificilmente precise ser alterada no futuro, mas deve ter métodos que facilite sua extensão para a modificação de seu comportamento e sem que seja necessário a modificação da classe base.

O princípio do Aberto/Fechado é um grande aliado dos desenvolvedores, principalmente na hora da manutenção dos sistemas, pois se temos um sistema em pleno funcionamento, nada mais correto de bloquearmos a alteração das classes já existentes e que estão cumprindo corretamente suas tarefas, se precisamos da alteração de um comportamento destas classes podemos apenas estende-la com o mecanismo de herança por exemplo.

Outra grande vantagem da adoção deste princípio é que não serão mais necessários fazer testes unitários (**BOX 2**) novamente nas classes já existentes, somente na classe que está sendo adicionada naquele momento.

BOX 2. Testes Unitários

Os Testes Unitários são um assunto vital para o bom desenvolvimento de qualquer sistema. Eles consistem em testes em cada unidade do sistema (classes), de modo que as mesmas estejam funcionando da melhor maneira. Estes testes unitários são bastante presente nas principais linguagens orientadas a objetos, porém quem programa em Delphi não costuma fazer estes testes, pois trabalham somente de forma procedural e não conseguem tirar proveito deste excelente princípio que previne a ocorrência de erros nas classes e tornam o código muito mais confiável.

LSP - Liskov Substituition Principle

O princípio da Substituição de Liskov leva este nome pois o seu conceito foi introduzido por Bárbara Liskov e Jeannette Wing em 1993 através do artigo Family Values: A Behavioral Notion of Subtyping.

Este princípio nos ensina que um objeto de uma classe qualquer deve permitir a substituição em um sistema por qualquer instância de suas classes filhas e classes derivadas também devem possibilitar a substituição por sua classe base.

Este princípio é vital para a boa aplicação da orientação a objetos, pois ele facilita e muito o uso do polimorfismo (**BOX 3**), pois desta forma todo e qualquer método pode ser utilizado tanto na classe base quanto em todas as especializações dela sem nenhum problema.

BOX 3. Polimorfismo

Em Orientação a Objetos o Polimorfismo permite que usemos referências de tipos de classes mais abstratas para representar o comportamento das classes concretas que referenciam. Assim é possível tratar várias classe de maneira homogênea através da interface da classe mais abstrata.

Se temos uma classe filha que não implementa todos os métodos da classe pai então há de se considerar que talvez estejamos fazendo um mal uso do mecanismo de herança em nossa modelagem de classes e que esta herança na realidade nem deveria existir.

Devemos ter muito cuidado com o uso da herança em nossos sistemas, pois a herança muitas vezes acaba quebrado o

encapsulamento de nossas classes ou expondo para as classes derivadas métodos e propriedades que nem fazem sentido para ela.

SRP (Sigle Responsibility Principle) na prática

Existem vários exemplos de implementação deste princípio na web, porém na sua grande maioria trazem abordagens muito abstratas sobre o assunto, que foge do dia a dia do desenvolvedor. Nós, porém iremos trazer um exemplo de aplicativo comum a todos os programadores Delphi, que é o uso de um cadastro de clientes, com persistência e log de erros.

Consideramos uma aplicação VCL Win32, com um cadastro de clientes simples, onde temos um formulário com botões para criarmos um novo cliente e outro para persistir este cliente na base de dados, neste caso um arquivo XML simples vinculado a um ClientDatSet, além do componente ApplicationEvents que nos proporciona a codificação de alguns eventos importantes da aplicação. Veja a interface com os componentes visuais e não visuais na **Figura 1**.

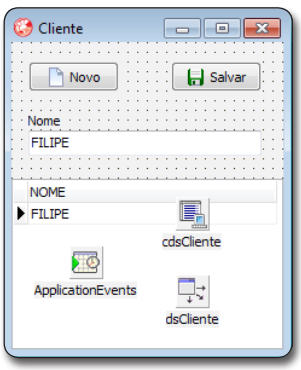


Figura 1. Interface do Cadastro de Clientes

Na **Listagem 1** a interface da classe do formulário, juntamente com a implementação dos eventos dos botões e do evento OnException do componente ApplicationEvents.

Vamos pensar neste formulário e código de acordo com o princípio de responsabilidade única, vemos que são muitas as responsabilidades presentes neles. Vemos que além de fazer a interface com o usuário que é a única responsabilidade que ele deveria ter, ele faz a persistência dos dados através do ClientDataSet (linhas 18 e 23).

Existe ainda uma terceira responsabilidade atribuída a esta classe de formulário que é a realização de log de erros da aplicação em um arquivo de texto no disco.

A implementação deste log se dá através da codificação do evento OnException do componente TApplicationEvents que nos passa por parâmetro qual a exceção levantada pela aplicação. Definimos uma constante do local de persistência do arquivo de erros (linha 2) e usamos uma variável do tipo TextFile para trabalharmos com esse arquivo (linha 3).

No corpo do método, fazemos a referência ao local do arquivo na variável criada (linha 5), e dentro de um bloco try/finally verificamos se o arquivo já existe no disco (linha 7), se sim apenas adicionamos uma linha no final do arquivo, senão criamos e adicionamos uma linha no final. Depois basta escrever o texto da mensagem de exceção no arquivo através do comando writeln.

Veja então que existem vários motivos para que modifiquemos esta classe, mudança do visual, mudança na persistência e mudança no log de erros, portanto esta classe fere completamente o princípio da responsabilidade única.

Listagem 1. Interface e Implementação do código do formulário de clientes

```
01 procedure TfrmCliente.ApplicationEventsException
     (Sender: TObject; E: Exception);
02 const ARQ_ERRO = 'C:\LogErros.txt';
03 var Arquivo: TextFile;
04 begin
05   AssignFile(Arquivo, ARQ_ERRO);
06   try
07     if FileExists(ARQ_ERRO) then
08       Append(Arquivo)
09     else Rewrite(Arquivo);
10     Writeln(Arquivo, E.Message);
11   finally
12     CloseFile(Arquivo);
13   end;
14 end;
15
16 procedure TfrmCliente.btnNovoClick(Sender: TObject);
17 begin
18   cdsCliente.Append;
19 end;
20
21 procedure TfrmCliente.btnSalvarClick(Sender: TObject);
22 begin
23   cdsCliente.Post;
24 end;
```

SRP (Sigle Responsibility Principle) refatorado

Agora vamos considerar novamente o exemplo do cadastro de clientes e vamos refatorar a aplicação de forma que ela possa seguir o princípio de responsabilidade única pregado nos princípios SOLID.

O primeiro passo que devemos tomar para refatorar a aplicação é o de mover o ClientDataSet responsável pela persistência dos dados para um Data Module chamado dmCliente. Esta prática já é bastante praticada por desenvolvedores Delphi, porém somente isso não é suficiente para termos um design de aplicação, se continuarmos acessando diretamente o ClientDataSet no nosso formulário (dmCliente.cdsCliente.Post) estaremos ferindo outro princípio dos mais básicos da orientação a objetos, que é o de encapsulamento, ou seja, uma classe ser a única a poder acessar e modificar seus atributos.

Neste caso o ClientDataSet presente no DataModule é um atributo do mesmo, sendo que o formulário não deve acessar diretamente este objeto, por isso necessitamos criar dois métodos públicos no DataModule chamados Novo e Salvar que sejam os responsáveis por chamar os métodos Append e Post do ClientDataSet, de maneira que agora estamos respeitando este princípio. Podemos ver esses novos métodos na **Listagem 2**.

Conforme visto, criamos os métodos necessários para o Append e o Post do ClientDataSet, porém quem nos garante que eles serão chamados no formulário? Quando adicionamos um componente, seja ele num formulário ou data module, o Delphi cria um objeto dentro desta classe na seção Published da mesma.

Outra boa prática de programação é a que diz que um atributo deve estar na seção private de uma classe, para que estes objetos não sejam acessados indevidamente por outros módulos do sistema. O ClientDataSet, ao criarmos suas colunas, cria objetos do tipo TField para um dos seus campos, portanto devemos mover tanto o ClientDataSet quanto os campos TField para a seção private do formulário (linhas 4 a 6), estando desta forma a respeitar o princípio.

Porém simplesmente movendo os objetos da seção published para a private não é o suficiente, a aplicação irá compilar normalmente, porém em tempo de execução termos um erro conforme **Figura 2**.

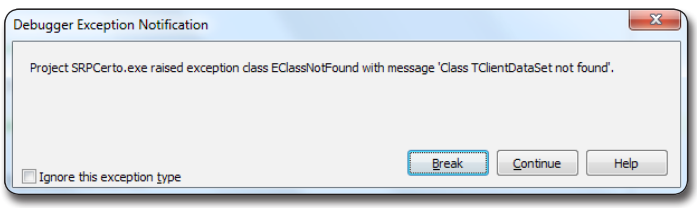


Figura 2. Erro ocorrido ao movermos componentes para a seção private

Este erro ocorre porque o sistema de streaming do Delphi precisa conhecer os nomes de todas as classes, afim de localizar todas as referências de classes necessárias para construir os componentes ao carregar o DFM, para contornar este problema, basta fazermos o registro das classes na seção initialization da unidade (linha 28).

Se executarmos o aplicativo, os eventos referentes ao componente cdsCliente não irão funcionar, devido ao ClientDataSet não ser inicializado e tais eventos estarem referenciando objetos nulos. O Delphi inicializa automaticamente somente os componentes presentes na seção published, para corrigirmos este problema, podemos no evento OnCreate do formulário fazer esta inicialização manualmente (linha 14).

Como último ajuste, o Delphi cria uma variável global para cada formulário e data module criados na própria unidade. Sabemos que variáveis globais são uma péssima prática de programação, por esse motivo removemos a declaração da variável global criada no dmCliente, agora ele não será automaticamente criado pelo Delphi e sim iremos instanciar e liberar o data module da memória quando necessário.

Agora sim temos um DataModule com seus objetos encapsulados e somente com métodos públicos para serem consumidos pelo formulário. Foram necessários alguns ajustes e algumas codificações extras que não são necessárias ao trabalharmos de forma procedural, porém vale o esforço para termos um desenvolvimento RAD e orientado a objetos ao mesmo tempo.

A próxima responsabilidade a ser extraída do formulário é a de Log de Erros. Para isso desenvolvemos uma nova classe chamada TLogErro, conforme **Listagem 3**.

Listagem 2. Interface e Implementação do DataModule de persistência

```
01 TdmCliente = class(TDataModule)
02   procedure DataModuleCreate(Sender: TObject);
03 private
04   cdsCliente: TClientDataSet;
05   cdsClienteID: TAutoIncField;
06   cdsClienteNOME: TStringField;
07 public
08   procedure Novo;
09   procedure Salvar;
10 end;
11
12 procedure TdmCliente.DataModuleCreate(Sender: TObject);
13 begin
14   cdsCliente := FindComponent('cdsCliente') as TClientDataSet;
15 end;
16
17 procedure TdmCliente.Novo;
18 begin
19   cdsCliente.Append;
20 end;
21
22 procedure TdmCliente.Salvar;
23 begin
24   cdsCliente.Post;
25 end;
26
27 initialization
28   RegisterClasses([TClientDataSet, TAutoIncField, TStringField]);
```

Listagem 3. Interface e Implementação da classe TLogErro

```
01 TLogErro = class
02 private
03   const ARQ_ERRO = 'C:\LogErros.txt';
04 public
05   procedure Gravar(E: Exception);
06 end;
07
08 procedure TLogErro.Gravar(E: Exception);
09 var
10   Arquivo: TextFile;
11 begin
12   AssignFile(Arquivo, ARQ_ERRO);
13   try
14     if FileExists(ARQ_ERRO) then
15       Append(Arquivo)
16     else Rewrite(Arquivo);
17     Writeln(Arquivo, E.Message);
18   finally
19     CloseFile(Arquivo);
20   end;
21 end;
```

Esta classe terá um único método público chamado Gravar, que irá receber por parâmetro a exceção gerada pela aplicação e fazer a operação de gravação no arquivo de log de erros do disco. Atente pelo fato da constante ARQ_ERRO estar na seção private da classe, esta é outra boa prática que podemos adotar, pois somente faz sentido esta constante para essa classe, de modo que esta constante não precisa estar acessível a outras unidades.

Iremos receber do formulário principal qual a exceção levantada pela aplicação para coletarmos posteriormente a mensagem de erro. A implementação do método Gravar segue a mesma ideia da codificação na listagem onde usamos no evento OnException do componente ApplicationEvents.

No formulário, não teremos mudanças na interface em relação aos componentes, somente o componente ClientDataSet foi movido para o DataModule.

Agora o formulário irá consumir os recursos das duas novas classes adicionada no projeto, a TdmCliente e TLogErro, para isso declaramos na seção private dois objetos destas classes e nos eventos OnCreate e OnDestroy fizemos a criação e destruição destes objetos respectivamente, conforme pode ser visto na **Listagem 4**.

Agora na implementação dos cliques dos botões Novo e Salvar, chamamos os métodos do criados no data module para a execução das funcionalidades. Já no evento OnException do componente ApplicationEvents, chamamos o método gravar da classe TLogErro passando o objeto que representa a exceção levantada pelo sistema.

Agora temos as responsabilidades divididas em três classes diferentes, de maneira que cada uma destas responsabilidades podem ser alteradas sem que haja impacto nas outras.

Listagem 4. Declaração objetos e implementação de eventos

```
01 private
02   LogErro: TLogErro;
03   dmCliente: TdmCliente;
04
05 procedure TfrmCliente.ApplicationEventsException
     (Sender: TObject; E: Exception);
06 begin
07   LogErro.Gravar(E);
08 end;
09
10 procedure TfrmCliente.btnNovoClick(Sender: TObject);
11 begin
12   dmCliente.Novo;
13 end;
14
15 procedure TfrmCliente.btnSalvarClick(Sender: TObject);
16 begin
17   dmCliente.Salvar;
18 end;
19
20 procedure TfrmCliente.FormCreate(Sender: TObject);
21 begin
22   LogErro := TLogErro.Create;
23   dmCliente := TdmCliente.Create(nil);
24 end;
25
26 procedure TfrmCliente.FormDestroy(Sender: TObject);
27 begin
28   LogErro.Free;
29   dmCliente.Free;
30 end;
```

OCP – Open Closed Principle na Prática

Este é um dos princípios de modelagem de classes mais polêmicos, desconhecido e pouco utilizado, que nos diz que uma classe deve estar aberta para extensão e fechada para alteração, mas o que isso significa na prática? Os softwares são evolutivos, raramente um software é desenvolvido uma única vez e nunca mais será modificado, mas onde este princípio tenta nos ajudas nesta evolução?

Para exemplificarmos o princípio do aberto/fechado na prática, agora vamos considerar uma classe de abstração de clientes de uma determinada empresa. Esta empresa concede descontos para compras à vista, porém estes descontos são diferenciados de acordo com a categoria do cliente. Esses clientes com o passar do tempo e de acordo com suas quantidades de compras e pagamentos adequados podem ser classificados como clientes ouro ou prata, de forma que possuem descontos especiais. Novos clientes ou clientes recentes são categorizados como normais e possuem um desconto padrão.

De acordo com o negócio, clientes normais possuem desconto de 5% nas compras à vista, já clientes prata possuem 10%, enquanto clientes ouro tem 20% de desconto.

A primeira codificação desta solução pode ser vista na **Listagem 5**, onde num primeiro momento somos levados a criar uma enumeração para as categorias de clientes (linha 2), onde temos se o cliente será ouro, prata ou normal, para posteriormente criarmos uma propriedade na classe cliente que identifique cada cliente em determinada categoria (linha 8).

Já para a concessão do desconto criamos um método público chamado GetDescontoAVista que recebe por parâmetro o total da venda feita para o cliente e devolvemos o valor atualizado com o desconto já calculado.

Para calcularmos o desconto fazemos uma série de testes condicionais onde verificamos qual a categoria do cliente e concedemos o desconto de acordo com o tipo de cliente.

Listagem 5. Primeira implementação da solução da categoria dos clientes e descontos

```
01 type
02   TCategoria = (cOuro, cPrata, cNormal);
03
04   TCliente = class
05   private
06     FCategoria: TCategoria;
07   public
08     property Categoria: TCategoria read FCategoria write FCategoria;
09     function GetDescontoAVista(ATotalVenda: Double): Double;
10   end;
11
12 implementation
13
14 { TCliente }
15
16 function TCliente.GetDescontoAVista(ATotalVenda: Double): Double;
17 begin
18   if FCategoria = cOuro then
19     Result := ATotalVenda * 0.2
20   else if FCategoria = cPrata then
21     Result := ATotalVenda * 0.1
22   else if FCategoria = cNormal then
23     Result := ATotalVenda * 0.05
24 end;
```

A princípio o código parece simples e conciso, mas o que acontece se adicionarmos mais uma categoria chamada Bronze, por exemplo? Bem, a solução seria adicionar mais um If na implementação do método GetDescontoAVista e estaria pronto.

Mas qual o problema de mais um If? Bem, a adição de mais um If indica que devemos substituir esta classe que já estava funcionando perfeitamente e testarmos novamente os descontos para cada tipo de cliente para vermos se esta adição não ocasionou nenhum bug na aplicação.

Nota

Em alguns sistemas encontramos uma enorme quantidade de IFs, o que torna o sistema de difícil entendimento e algumas vezes bem confusos. Muitos desses IFs podem ser substituídos por soluções muito mais elegantes como o uso de herança e polimorfismo para alterar o comportamento de um método. Excessos de IFs são sintomas de classes mal estruturadas e que não seguem as boas práticas de orientação a objetos.

OCP – Open Closed Principle Refatorado

O princípio nos indica que as classes devem estar fechadas para modificação, portanto uma nova categoria não deveria ocasionar uma modificação na classe cliente. O princípio também diz que as classes devem estar abertas para extensão, para que possamos herdar da mesma e apenas modificar o comportamento dela, sem que precisamos recompilar e substituir a classe base no sistema.

Para uma categoria não afetar de nenhuma maneira a classe atual que se encontra em perfeito funcionamento em ambiente de produção, devemos criar uma especialização da mesma e sobrescrever o método GetDescontoAVista.

Podemos verifica na **Listagem 6** a classe cliente com o método GetDescontoAVista com a diretiva virtual no final de sua declaração (linha 3), possibilitando assim que possamos criar classes descendentes e estender este método modificando seu comportamento ou apenas adicionando mais algum recurso.

Veja que agora temos mais duas classes criadas, TClienteOuro e TClientePrata, e cada uma delas sobrescreve o método GetDescontoAVista da classe TCliente através da diretiva override, e implantando o desconto de acordo com a categoria do clientes.

Agora se tivéssemos mais uma categoria de cliente bronze, por exemplo, bastaria criarmos mais uma classe descendente de TCliente sobrescrevendo novamente o método GetDescontoAVista e aplicando o novo desconto.

Observe que a criação de uma nova categoria não iria impactar em nada as categorias anteriores e nenhum teste seria necessário nas classes TCliente, TClienteOuro e TClientePrata, somente a TClienteBronze precisaria ser testada e teríamos a garantia de que nenhum bug seria adicionado as categorias já existentes, na prática é isto que prega o princípio do aberto/fechado.

Desenvolver classes propensas a extensão é uma das chaves da orientação a objetos, quando uma nova funcionalidade ou comportamento precisa ser adicionado no sistema, é esperado que estas classes existentes sejam estendidas e não alteradas, de modo que o código original delas permaneça intacto e confiável, com as novas funcionalidades são implementadas através da extensibilidade.

Quando estamos aprendendo a orientação a objetos ouvimos muito falar sobre abstração, e ela permite que este princípio funcione adequadamente, se um software possui abstrações bem definidas, logo ele estará aberto para extensão.

Criar códigos extensíveis é uma tarefa desempenhada principalmente por desenvolvedores maduros, que conseguem antever a evolução do sistema e criar suas classes habilitadas para este fim, com uma arquitetura e design duradouros, tornando os sistemas de ótima qualidade e manutenibilidade.

Para o uso desde princípio, muitos desenvolvedores usam o padrão de projeto Strategy (**BOX 4**).

Listagem 6. Classe Cliente refatorada para seguir o princípio do aberto/fechado

```
01 type
02  TCliente = class
03    function GetDescontoAVista(ATotalVenda: Double): Double; virtual;
04  end;
05  TClienteOuro = class(TCliente)
06    function GetDescontoAVista(ATotalVenda: Double): Double; override;
07  end;
08  TClientePrata = class(TCliente)
09    function GetDescontoAVista(ATotalVenda: Double): Double; override;
10  end;
11
12 implementation
13
14 { TCliente }
15 function TCliente.GetDescontoAVista(ATotalVenda: Double): Double;
16 begin
17   Result := ATotalVenda * 0.05;
18 end;
19
20 { TClienteOuro }
21 function TClienteOuro.GetDescontoAVista(ATotalVenda: Double): Double;
22 begin
23   Result := ATotalVenda * 0.2;
24 end;
25
26 { TClientePrata }
27 function TClientePrata.GetDescontoAVista(ATotalVenda: Double): Double;
28 begin
29   Result := ATotalVenda * 0.1;
30 end;
```

BOX 4. Strategy

A lógica condicional é uma das estruturas mais complexas e utilizadas no desenvolvimento de softwares corporativos. Lógicas condicionais tendem a crescer e tornar-se cada vez mais sofisticadas, maiores e mais difíceis de manter com o passar do tempo. O padrão Strategy ajuda a gerenciar toda essa complexidade imposta pelas lógicas condicionais. O Padrão Strategy sugere que se produza uma família de classes para cada variação do algoritmo e que se forneça para a classe hospedeira uma instância de Strategy para a qual ela delegará em tempo de execução. Um dos pré-requisitos para o Strategy é uma estrutura de herança onde cada subclasse específica contém uma variação do algoritmo. Assim, o padrão Strategy possui diversos benefícios como clarificar algoritmos ao diminuir ou remover lógica condicional, simplificar uma classe ao mover variações de um algoritmo para uma hierarquia, e habilitar um algoritmo para ser substituído por outro em tempo de execução.

Em resumo o padrão Strategy pode ser utilizado quando se tem as seguintes situações:

Quando muitas classes relacionadas diferem apenas no seu comportamento;

Quando necessita-se de variantes de um algoritmo;

Quando se precisa ocultar do usuário a exposição das estruturas de dados complexas, específicas do algoritmo;

Quando uma classe define muitos comportamentos e por sua vez eles aparecem como diversos "IFs". Com isso esses comandos condicionais são movidos para sua própria classe Strategy.

LSP - Liskov Substituition Principle na Prática

Este princípio em resumo diz que: se uma classe B é um subtipo de uma classe A, então objetos da classe A podem ser substituídos por objetos da classe B.

Imaginemos primeiramente um exemplo bem abstrato deste princípio. Temos uma classe TVoyage que herda da classe TCarro. A classe TVoyage possui um método chamado Andar, porém este método deve também existir na classe TCarro, para que possamos a qualquer momento em outro módulo do sistema, fazermos referência a classe TVoyage como sendo uma TCarro, ou seja, devemos ter um método abstrato chamado Andar na classe TCarro, sendo que a classe TVoyage o implementa. Desta forma podemos acessar um objeto de TVoyage através de uma referência TCarro.

Para exemplificarmos o princípio vamos considerar um problema bastante comum no dia-a-dia do desenvolvedor, que é a geração de arquivos nos mais diversos formatos. Frequentemente somos requisitados para geração de vários tipos de arquivo como Texto, Excel, Pdf, etc.

O que ocorre geralmente é a criação de uma classe base com o nome do arquivo e o texto a ser gerado e uma classe filha com as implementações de cada tipo de arquivo, conforme pode ser visto na **Listagem 7**.

Normalmente também se cria uma classe apenas para fazer todas as gerações e facilitar o uso por parte dos clientes. Criamos portanto a classe TGeradorArquivo que possui um único método chamado GerarArquivo que recebe uma classe Arquivo por parâmetro, que deverá ser feita a geração de acordo com o tipo de classe passada.

No corpo do método temos testes condicionais onde verificamos o tipo da classe passada através do operador IS e chamamos o método apropriado para cada tipo de classe, através do uso de Typecast com o operador AS.

Existe grandes problemas na modelagem destas classes, esses métodos de geração não deveriam ser específicos de cada classe e sim estar na classe base pré-definido para usufruirmos do poder do polimorfismo na classe de geração de arquivos.

Você deve pensar neste momento, mas as formas de geração de arquivos Texto e Pdf são completamente diferentes. Está correto, elas são mesmo completamente diferentes, mas sua implementação que é diferente, a interface de geração é a mesma, isso que importa no momento da definição.

Uma das principais regras da orientação a objetos é que devemos programar sempre voltados a interface, não para a implementação. É muito mais importante definir o que uma classe vai fazer do como ela irá fazer.

Com esta abordagem, no momento que tivermos a geração de um novo tipo de arquivo pelo sistema, teremos que adicionar mais um else if, o que já vimos não ser uma boa prática de orientação a objetos. Assim podemos afirmar que as classes TArquivoTexto e TArquivoPdf não estão em conformidade com o princípio de Substituição de Liskov, uma vez que não podemos substituí-la pela classe Arquivo na classe TGeracaoArquivo, porque os métodos de geração são específicos de cada classe derivada. Sendo assim somos obrigados a verificar o tipo e fazer typecast para fazer a chamada do método apropriado.

Listagem 7. Implementação defeituosa da geração de arquivos ferindo o princípio LSP

```
01 type
02  TArquivo = class
03  protected
04    FNome: string;
05    FTexto: string;
06  public
07    property Nome: string read FNome write FNome;
08    property Texto: string read FTexto write FTexto;
09  end;
10
11  TArquivoTexto = class(TArquivo)
12    procedure GerarArquivoTexto;
13  end;
14
15  TArquivoPdf = class(TArquivo)
16    procedure GerarArquivoPdf;
17  end;
18
19  TGeradorArquivo = class
20    procedure GerarArquivo(AArquivo: TArquivo);
21  end;
22
23 procedure TGeradorArquivo.GerarArquivo(AArquivo: TArquivo);
24 begin
25  if AArquivo is TArquivoTexto then
26    (AArquivo as TArquivoTexto).GerarArquivoTexto()
27  else if AArquivo is TArquivoPdf then
28    (AArquivo as TArquivoPdf).GerarArquivoPdf();
29 end;
```

LSP - Liskov Substitution Principle Refatorado

Uma regra geral para atingirmos o objetivo de respeitar este princípio da orientação a objetos é o de nunca termos métodos específicos em nossas classes descendentes. Todo método necessário numa classe filha da classe pai, deve obrigatoriamente ser previsto na classe pai, mesmo que seja apenas um protótipo, com o uso da diretiva abstract, onde não precisamos fazer nenhuma implementação do método, simplesmente indicamos que este método deve ser implementado pelas classes descendentes.

Esta medida é muito inteligente e nos possibilita fazer uso do polimorfismo para fazermos referência a qualquer classe de uma hierarquia de classes através da classe base da mesma.

Na **Listagem 8** temos a definição das classes TArquivo, TArquivoTxt e TArquivoPdf, seguindo o princípio de substituição de Liskov. Veja que não será mais necessário o uso do typecast para fazer a geração correta dos arquivos.

O primeiro passo feito na refatoração destas classes é o de criarmos um protótipo do método de geração de arquivos na classe base (linha 9), conseguimos isto através das diretivas virtual e abstract, fazendo com que não tenhamos que implementar este método na classe pai, apenas estamos indicando para todas suas descendentes que elas devem obrigatoriamente implementar este método de acordo com a sua forma de geração.

Nas classes descendentes, agora não temos mais um método com

cada nome e sim fazemos a sobrescrita do método Gerar da classe base com a codificação específica de cada tipo de geração.

Toda esta refatoração nos remete a um método GerarArquivo da classe TGeradorArquivo muito mais simples e sem o uso de type-cast (linha 30) que é propício a erros em tempo de execução.

Esta maneira de resolver este problema é muito mais elegante e aproveita todo o poder do polimorfismo para a geração de arquivos.

Este princípio nos alerta para o cuidado que devemos ter com nossas hierarquias de classes e é muito importante, pois nos sem a aplicação do mesmo a hierarquia de classe fica muito bagunçada e os testes unitários para uma classe base nunca teriam sucesso para a subclasse. Para evitarmos a violação deste princípio pode-se também aplicar o uso de padrões de projeto, como o composite (**BOX 5**), onde evitamos o uso da herança em decorrência da composição de classes.

Se conseguirmos seguir todos estes princípios apresentados neste artigo, teremos um código limpo, de fácil manutenção e muito mais propício à evolução. Mudanças no sistema serão de fácil acesso e suas correções serão em lugares bem específicos, e problemas não serão propagados para o restante do sistema.

O problema é que não é fácil escrever um código sólido e que siga tais técnicas, isto vai se conseguindo com experiência no desenvolvimento de software e estudo. O interessante é termos sempre classes pequenas com métodos enxutos, o que nos traz muita modularidade.

Listagem 8. Geração de arquivos obedecendo o princípio LSP

```
01 type
02  TArquivo = class abstract
03  protected
04    FNome: string;
05    FTexto: string;
06  public
07    property Nome: string read FNome write FNome;
08    property Texto: string read FTexto write FTexto;
09    procedure Gerar; virtual; abstract;
10  end;
11
12  TArquivoTexto = class(TArquivo)
13    procedure Gerar; override;
14  end;
15
16  TArquivoPdf = class(TArquivo)
17    procedure Gerar; override;
18  end;
19
20  TGeradorArquivo = class
21    procedure GerarArquivo(AArquivo: TArquivo);
22  end;
23
24 implementation
25
26 { TGeradorArquivo }
27
28 procedure TGeradorArquivo.GerarArquivo(AArquivo: TArquivo);
29 begin
30  AArquivo.Gerar();
31 end;
```


DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

BOX 5. Composite

O padrão Composite é utilizado quando queremos reutilizar método e funções de algum objeto e colocar em qualquer outro, sendo vários objetos ligados a um único. Em certas estruturas necessitamos por muitas vezes de métodos e funções que já estão especificadas em um determinado objeto, sendo necessária apenas a ligação entre os dois para assim poder ser reutilizado o código. Este padrão pode ser usado sempre que desejarmos construir objetos, existindo objetos do mesmo tipo.

Participantes:

Componente:

* Declara a interface para objetos na composição;
* Implementa comportamento default para interface comum a todas as classes, como apropriado;
* Declara uma interface para acessar ou gerenciar seus componentes filhos;

Folha:

*Representa objetos folhas na composição. Uma folha não tem filhos;
*Define comportamento para objetos primitivos na composição.

Composição:

* Define comportamento para Componentes que têm filhos;
* Armazena Componentes filhos;
* Implementa operações relacionadas com filhos na interface do Componente.

Cliente:

*Manipula objetos na composição através da interface Componente.

Com o uso do padrão Composite podemos criar objetos com uma grande complexidade e eles serem compostos por outros objetos menores, além de deixar o código bem estruturado e de fácil entendimento, sendo rápida a forma de adicionar novos componentes, métodos e funções.

O Delphi é uma excelente ferramenta RAD onde é muito fácil sair arrastando vários componentes, fazendo várias referências muito rapidamente. Essas ligações e vínculos criados entre os componentes e os formulários e DataModules vão criando um grande acoplamento entre os módulos do sistema, de forma que sua expansão e manutenção propagam mudanças em cascata para todo o sistema, ou pior, causando bugs em partes do sistema já testadas e em funcionamento.

Muitos hão de perguntar, mas desenvolver orientado a objetos em Delphi é muito mais demorado, vale a pena? A resposta é sim! Vale muito mais a pena perder um tempo a mais num bom design de software na hora de sua criação, do que perder muito mais tempo depois em cada nova manutenção que se faz necessário.

Como já mencionado um software não se desenvolve do dia pra noite e acabou, não precisamos mais botar a mão. Já foi provado através das pesquisas que a grande maioria do tempo dos desenvolvedores é gasto em cima de manutenções e não em desenvolvimento de novos recursos, por isso todo o tempo que for necessário para desenvolver um software com boas práticas de programação e orientação a objetos é muito bem gasto e nos trará um grande benefício no futuro.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/

Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi (Desktop) e C# (Web), utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Links:

Delphi XE6 Trial
http://www.embarcadero.com/br/products/delphi

Artigos POO Uncle Bob
http://butunclebob.com/ArticleS.UncleBob

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional.
Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software.
Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviçoes); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DEVMEDIA

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486